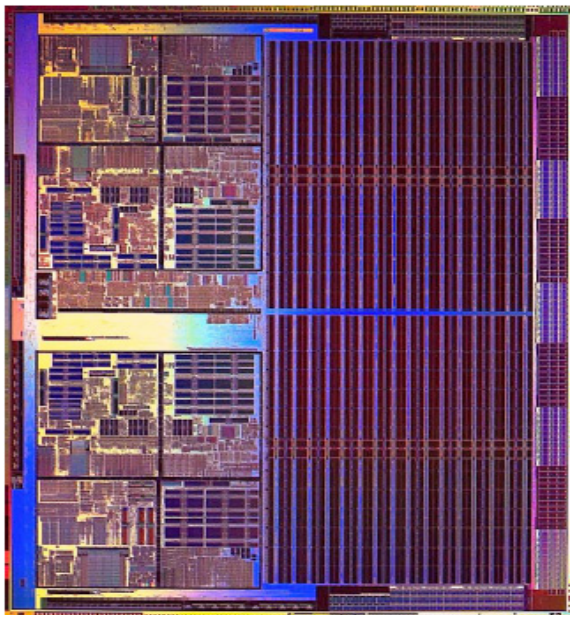
Developing Linux Applications on AMD Opteron™ and Athlon™ 64 Processors

AMD India Developer's Conference
Bangalore, 10-May-2006

David O'Brien
Senior Systems Software Engineer
Advanced Micro Devices, Inc.



Agenda

This talk is for application developers.

It's about maximizing application performance.

- AMD64 Linux Development Tools
- AMD64 Programmer's View
- Linux (& UNIX®) AMD64 Application Model
- AMD64 Performance Optimizing
- Linux AMD64 Gotcha's
- Memory & Multi-Processor Performance on AMD64
- NUMA Support in the Linux Kernel for AMD64
- 32-bit Apps on a 64-bit Linux running on AMD64

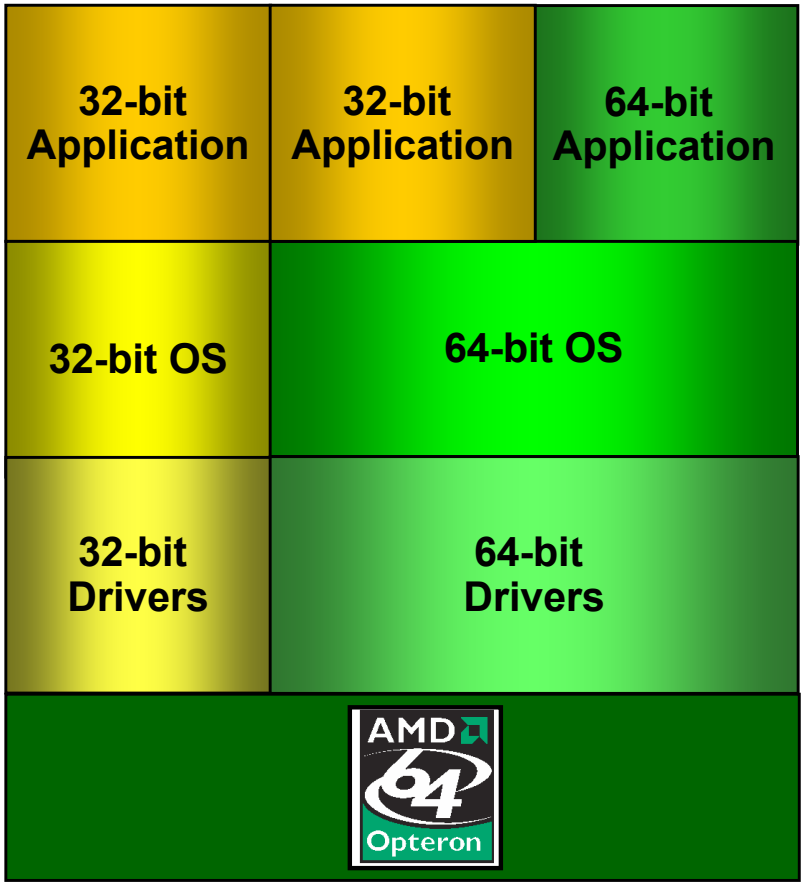
AMD64 Computing Strategy - April 2003

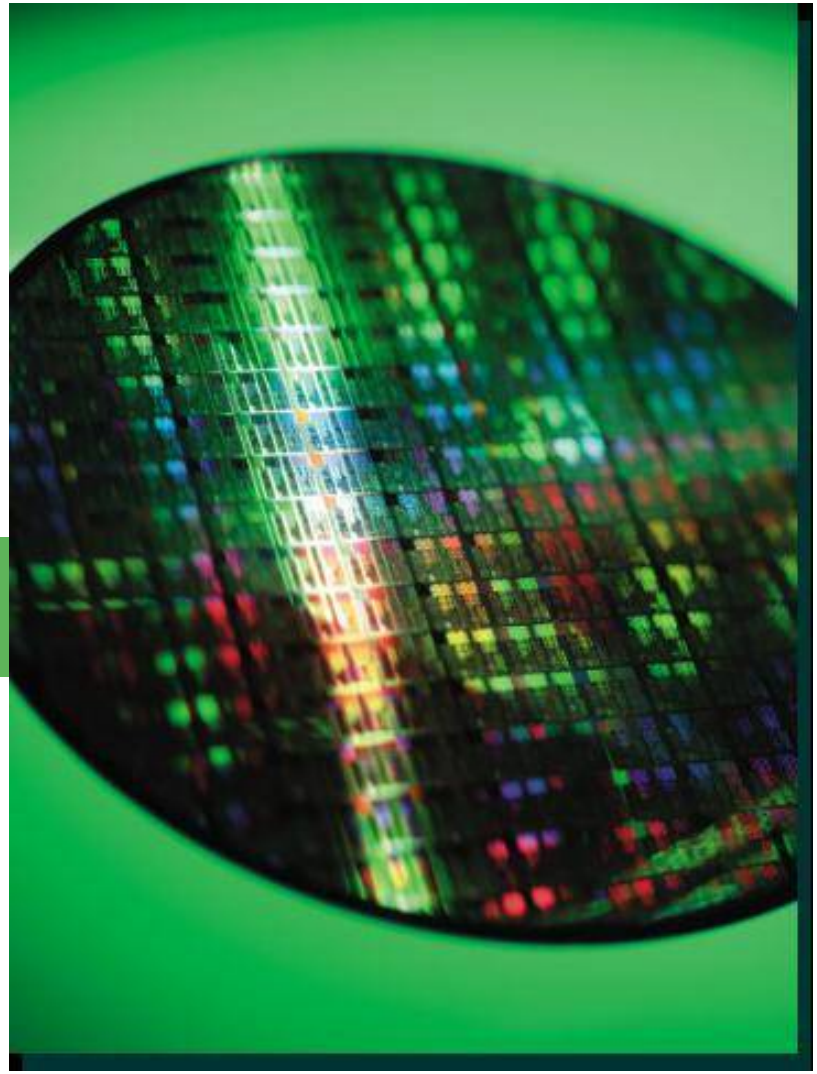
AMD took the x86 architecture and extended it to 64-bits to make the AMD64 architecture

Practical approach to introducing new capabilities

Extensions so simple and compatible, that the processor can support both x86 32-bit and AMD64 at full speed & performance

- Compatibility of 32-bit applications retained at full speed, without decreasing performance.
- Can move to 64-bit addressing and data types without giving up 32-bit compatibility (long mode)
- 64-bit applications can be developed using familiar techniques and tools (only two new instructions were added to support 64-bit addressing)
- Leverages the existing key PC infrastructure rather than needing to re-invent it.





AMD64 Linux Development Tools

Linux x86-64 Tools

- Many of the tools in the GNU C Compiler suite (and Binutils) have new switches for AMD64 use, so that one tool can compile 32-bit or 64-bit code.
- Generic architecture known as x86-64 (or "`x86_64`")
- Some of these tools can not handle both modes in a single application binary, and two different binaries are provided.
- "gcc" - GNU C compiler.
  "g++" - GNU C++ compiler.
  "g77" - GNU Fortran77 compiler.
  - Handles 32-bit and 64-bit code.
  - "`-m64`": Default, compile to 64-bit, AMD64 code.
  - "`-m32`": compile to 32-bit, x86 code.

Linux x86-64 Tools *(cont)*

- "as" - GNU assembler.
  - Handles 32-bit and 64-bit code.
  - "`--64`": Default, assemble AMD64 assembly into 64-bit code.
  - "`--32`": assemble i386 assembly into 32-bit code.
- ld - GNU linker.
  - Handles 32-bit and 64-bit code.
  - Invoking directly is discouraged; linking should be performed thru gcc/g++/g77.
  - Default: "`-m elf_x86_64`", produce AMD64 64-bit binaries.
  - "`-m elf_i386`": Produce 32-bit i386 binary.
- ar - GNU archive program.
  - Produces 32-bit and 64-bit libraries, depending on what files are passed into it.
  - Libraries containing both 32-bit and 64-bit code must not be created.

Linux x86-64 Tools *(cont)*

- nm - GNU program to list symbols in an object file.
  - Handles 32-bit & 64-bit object files w/o needing additional flags.
- gdb - GNU symbolic debugger.
  - For 64-bit AMD64 binaries only.
- gdb32 - GNU symbolic debugger.
  - For 32-bit i386 binaries only.
- strace - Captures list of systems calls
  - made by 64-bit AMD64 app.
- strace32 - Captures list of systems calls
  - made by 32-bit x86 app.
- ltrace - Captures list of library function calls
  - made by 64-bit AMD64 app.
  - No corresponding tool for 32-bit apps.

Linux x86-64 Tools *(cont)*

- objdump - Dumps object files.
  - Handles 32-bit & 64-bit object files w/o needing additional flags.
  - Examines ELF header to determine architecture.
- linux32 <app> - Sets personality to "i686".
  - All children of the app will also inherit the personality.
  - Sets up 3GB address space like for 32-bit Linux
- uname -m
  - Usually returns the normal "linux" personality "x86_64" -- even for 32-bit binaries.
  - However, some shell scripts check the architecture looking for "i686" and will not work when they see "x86_64".
  - This can be forced by doing "linux32 <shell_script>"

Useful GCC switches

- **-m32**
  - "-m32": compile to 32-bit, x86 code.
- **-m64**
  - "-m64": Default, compile to 64-bit, AMD64 code.
- **-O2**
  - GCC performs nearly all supported opts that do not involve a space-speed trade off.
  - Turns on all opts except for loop unrolling, function inlining, and register renaming.
- **-O3**
  - -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions and -frename-registers options.
- **-O0 -S**
  - Do not optimize, and generate assembly output.
- **-fprofile-generate** & **-fprofile-use ("PGO")**
  - Must use `-fprofile-generate` when compiling AND linking.
  - Implies `-fbranch-probabilities -fvpt -funroll-loops -fpeel-loops -ftracer`

Useful GCC switches for x86-64 platform

- -Wstrict-prototypes
  - Finds prototype and 64-bit problems.
- -mcmodel=kernel
  - Not documented in man page. Necessary switch for building kernel modules, they will crash upon loading without it.
- -fPIC
  - See section on PIC addressing
- `__amd64__` and `__linux__` is defined for writing conditional code.
- "lint" with:

```
gcc -Werror -Wall -W -Wstrict-prototypes
   -Wmissing-prototypes -Wpointer-arith -Wreturn-type
   -Wcast-qual -Wwrite-strings -Wswitch -Wshadow
   -Wcast-align -Wuninitialized -ansi -pedantic
   -Wbad-function-cast -Wchar-subscripts -Winline
   -Wnested-externs -Wredundant-decl
```

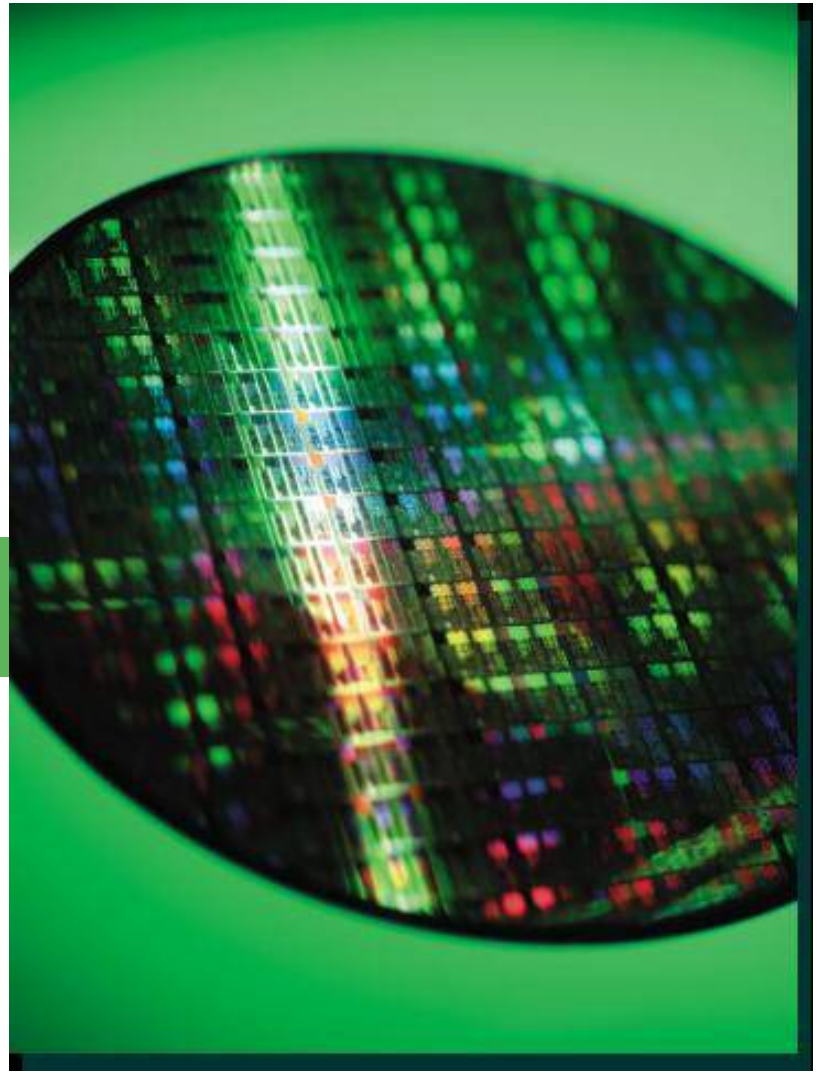
AMD64 Developer Tools

- Linux GNU compilers: gcc, g++, g77, gcj
  - GCC 3.3 / GCC 4 - optimized 64-bit

The Portland Group
Compiler Technology

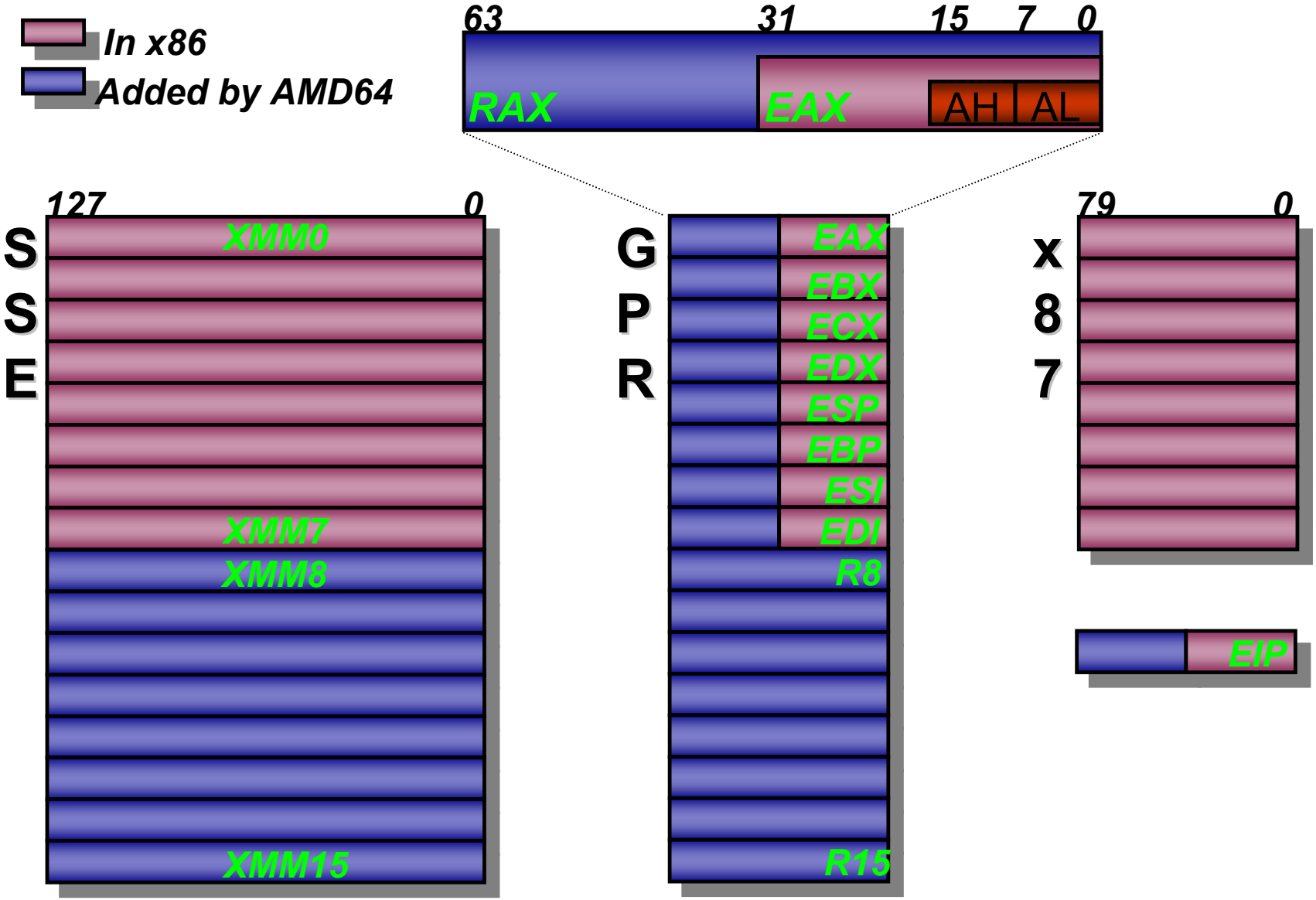
STMicroelectronics
More Intelligent Solutions

- PGI Workstation 6.1
  - Optimized C/C++, Fortran 77/90
  - For 64-bit & 32-bit Linux, 64-bit & 32-bit MS-Windows
  - OpenMP support; Full parallel debugging support
  - Performance goals are to be better than Intel compilers



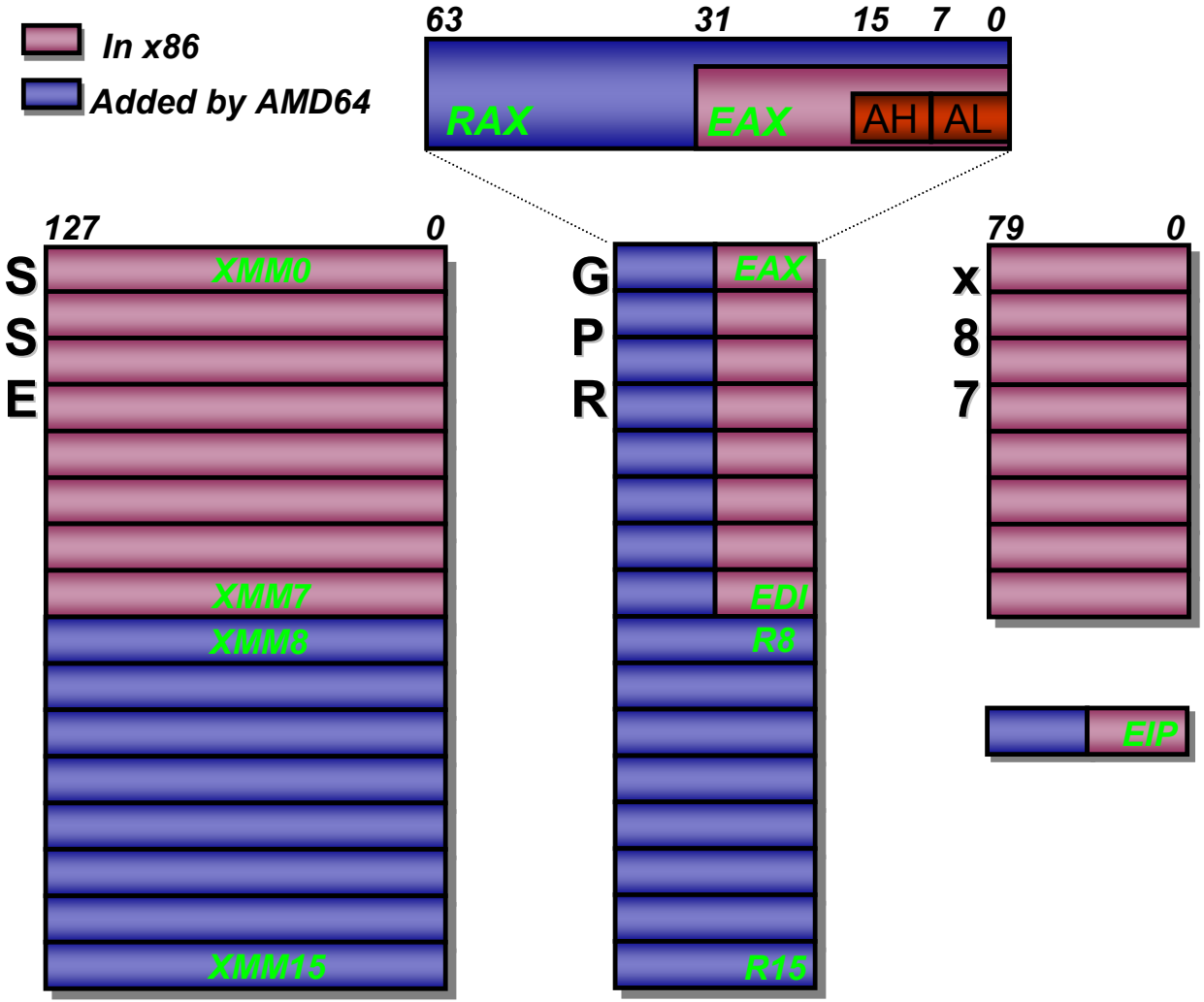
AMD64 Programmer's View

The x86-64 Programmer's Model



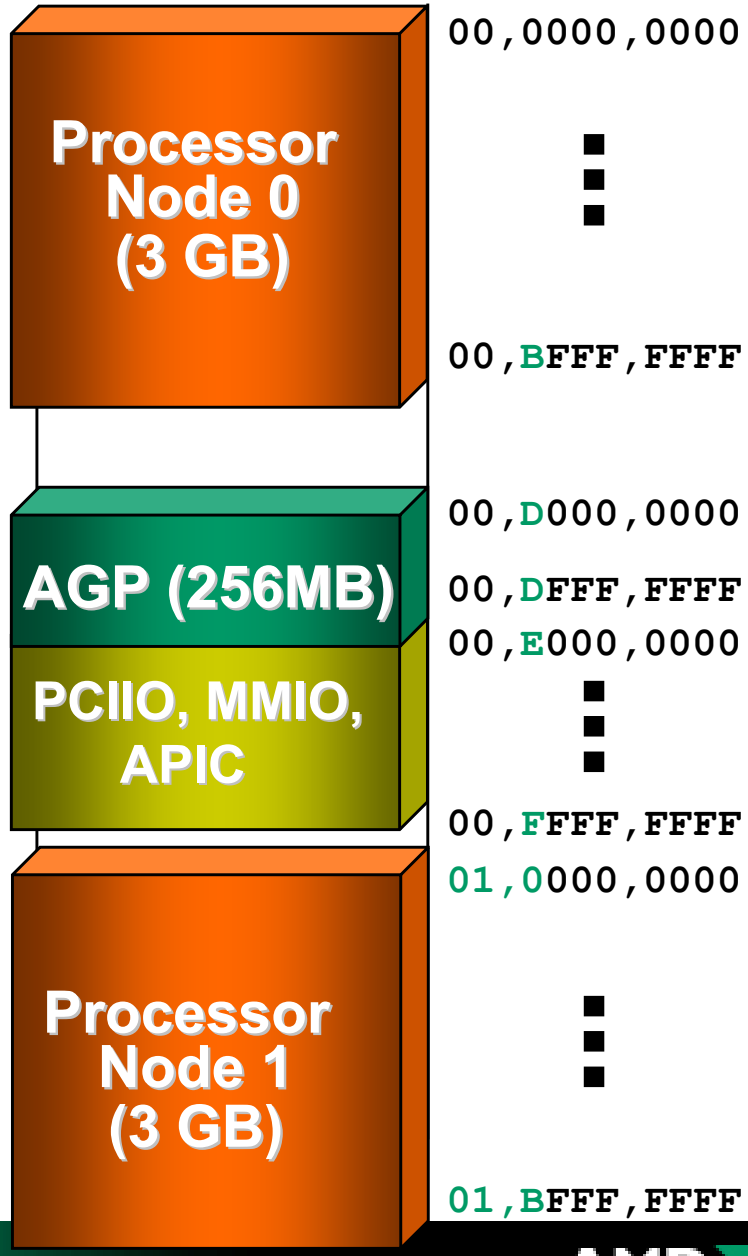
The AMD64 Programmer's Model

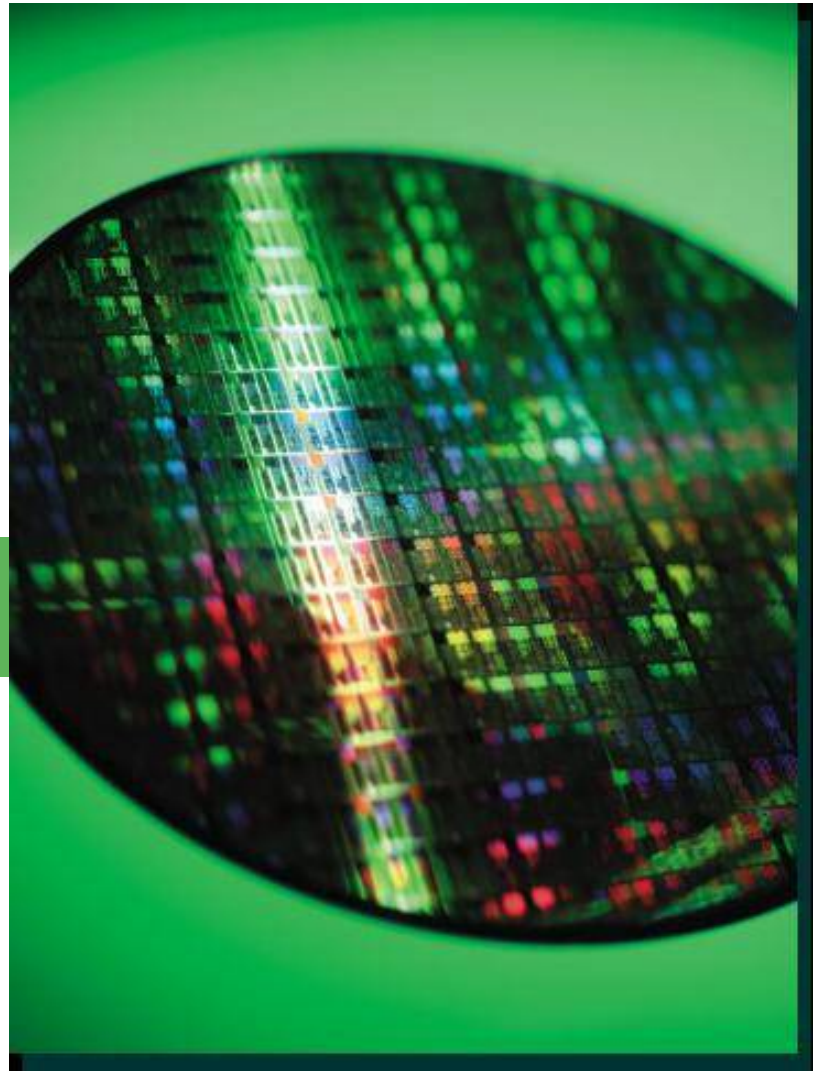
- 64-bit integer registers
- 64-bit Virtual Address
- 52-bit Physical Address
- Sixteen 64-bit integer regs
- Sixteen 128-bit SSE regs
- SSE3/SSE2/SSE Instruction Set
- Double precision scalar and vector operations
- 16x8, 8x16 way vector packed integer operations



64-bit Physical Memory Map Layout

- Assume
  - 2-CPU 6GB WS, 3GB DRAM per CPU
  - Non-node interleaved DRAM map
- BIOS maps these below 4GB so legacy OS & PCI devices can access:
  - AGP Aperture, PCI I/O-Map region, PCI Mem-Map region, APIC
- Simple mapping places 2nd CPU's memory above 4GB:
  - Other optimizations possible.





Linux (& UNIX®) AMD64 Application Model

ISO C/C++ Data Types (LP64 vs. LLP64)

Data Type	Linux/Unix ELF Size in bytes and alignment	MS-Windows64 Size in bytes and alignment
`char`	1	1
`short`	2	2
`int, enum`	4	4
long	***8***	***4***
`long long`	8	8
`pdiff_t, [u]intptr_t`	8	8
`size_t`	8	8
`float`	4	4
`double`	8	8
long double	***16***	***8***

Linux/ELF Calling Convention

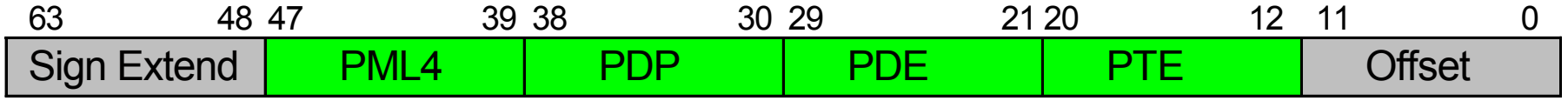
Register	*Status*	*Use*
`%rax`	Volatile	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register
`%rbx`	Non-volatile	optionally used as base pointer, must be preserved by callee
`%rdi, %rsi, %rdx, %rcx, %r8, %r9`	Volatile	Used to pass 1st, 2nd, 3rd, 4th, 5th, 6th integer args
`%rsp`	Non-volatile	stack pointer
`%rbp`	Non-volatile	optionally used as frame pointer, must be preserved by callee
`%r10`	Volatile	temporary register, used for passing a function's static chain pointer
`%r11`	Volatile	temporary register
`%r12-r15`	Non-volatile	Must be preserved by callee
`%xmm0-%xmm1`	Volatile	Used to pass and return floating point arguments
`%xmm2-%xmm7`	Volatile	used to pass floating point arguments
`%xmm8-%xmm15`	Volatile	temporary registers
`%mmx0-%mmx7`	Volatile	temporary registers
`%st0`	Volatile	temporary register; used to return long double arguments
`%st1-%st7`	Volatile	temporary registers
`%fs`	Volatile	Reserved for system use as thread specific data register

WARNING: The Microsoft Win64 calling convention is similar - but different register assignments.

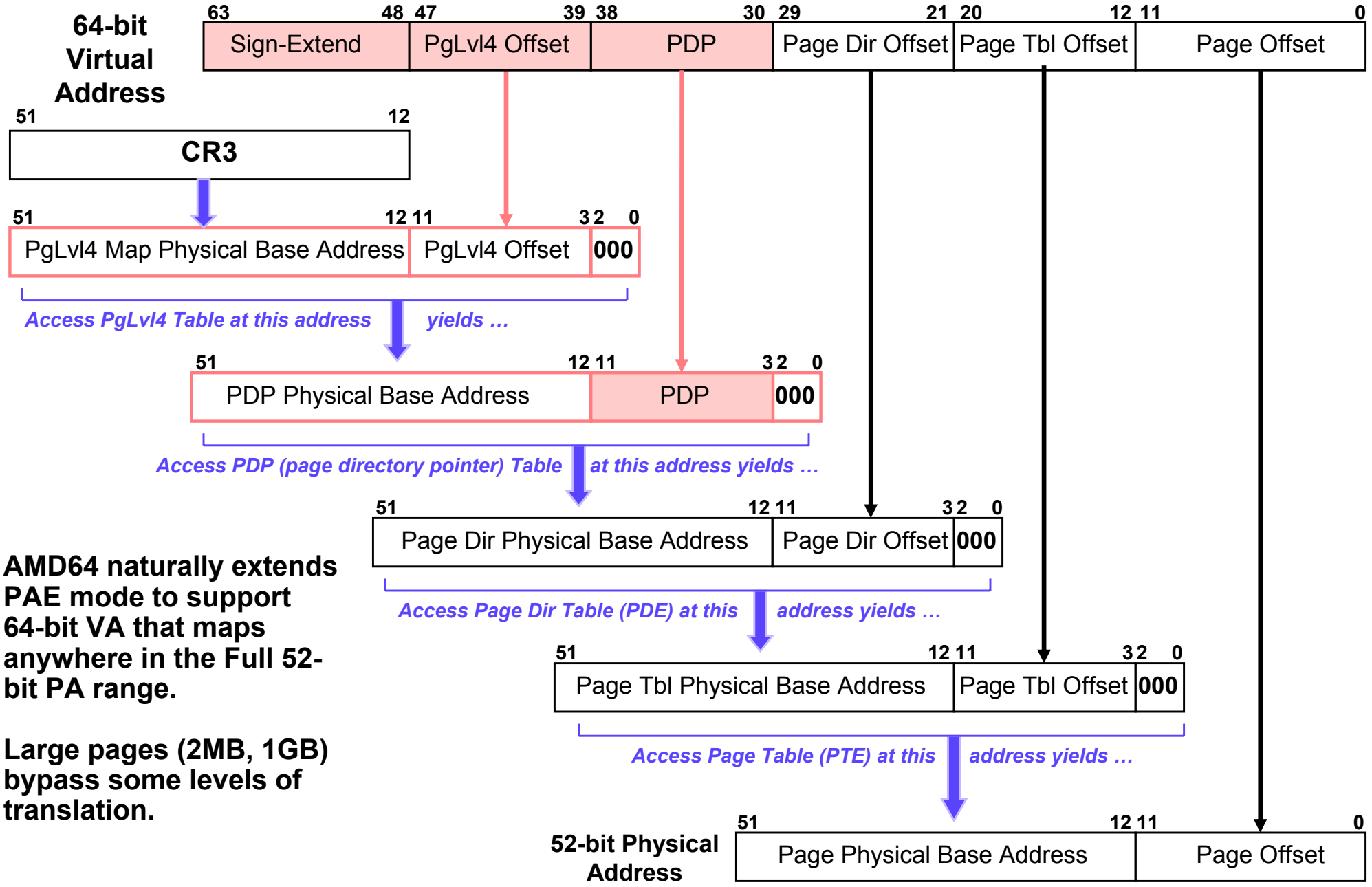
Page Tables

- New level added to paging structures: PL4
- PDP (page directory pointer) table fully expanded
  - From 4 entries (36-bit PA) to 512 entries (leading to 52-bit PA with PML4)
- PAE always enabled in Long Mode
  - leverage existing mechanism that extend page map entries to handle 64-bit addresses.
- Uses the same PAE format for table entries as 32-bit PAE mode.

The 48-bit virtual address is broken up into 4 fields to index into the 4-level paging structure.



AMD64 64-bit Virtual Addressing



AMD64 naturally extends PAE mode to support 64-bit VA that maps anywhere in the Full 52-bit PA range.

Large pages (2MB, 1GB) bypass some levels of translation.

Linux Virtual Address Space Mapping

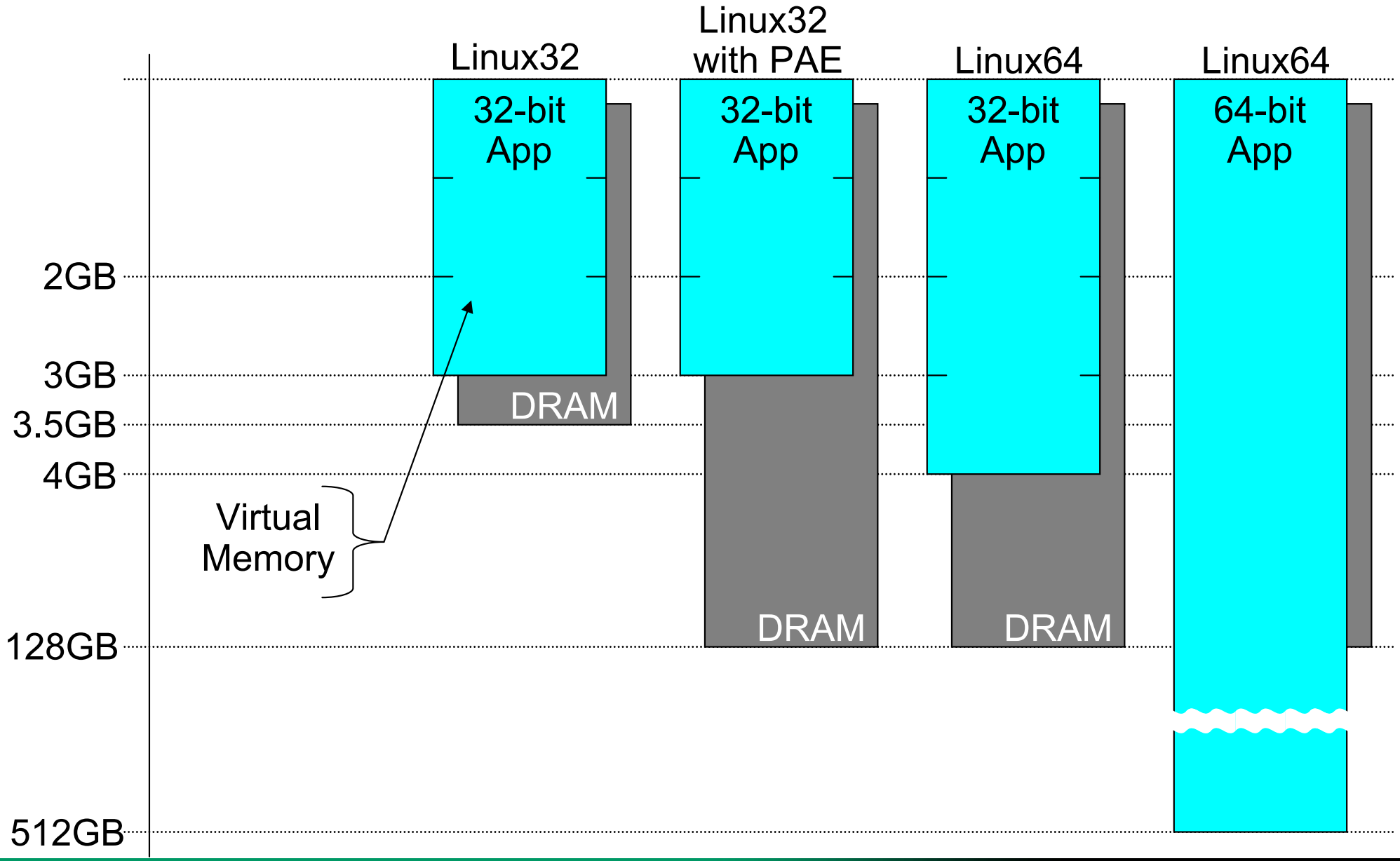
- Each PML4 slot is 512GB of Virtual Address Space.
  - There are 39 address bits [38:0] in the 3-level page table
- The User thread gets 512GB (aka PML4 slot 0).
  - But shared mappings start at 0x0000,002A,9555,6000
  - Usually somewhat less because shared mappings start at (512GB)/3.
- The kernel keeps all the rest for itself.
  - Kernel supports upto 507 PML4 slots for ~ 253TB VA.
- 32-bit Apps constrained to ~4GB address space

Linux Virtual Address Space Mapping (cont)

0000,0000,0000,0000 - 0000,007F,FFFF,FFFF:	user range
0000,0080,0000,0000 - 0000,00FF,0000,0000:	reserved to catch bad ptrs
0000,0100,0000,0000 - 0000,78ff,ffff,ffff:	~120TB map of Phys Mem
0000,7900,0000,0000 - ffff,fd00,ffff,ffff:	reserved
ffff,fe00,0000,0000 - ffff,feff,ffff,ffff:	pci mappings
ffff,ff00,0000,0000 - ffff,ff7f,ffff,ffff:	vmalloc/ioremap area
ffff,ffff,8000,0000 - ffff,ffff,8fff,ffff:	Kernel text
ffff,ffff,a000,0000 - ffff,ffff,ff5f,ffff:	loadable modules
ffff,ffff,ff60,0000 - ffff,ffff,ffdf,ffff:	vsyscalls

Note some items have to be w/in 32-bit signed range

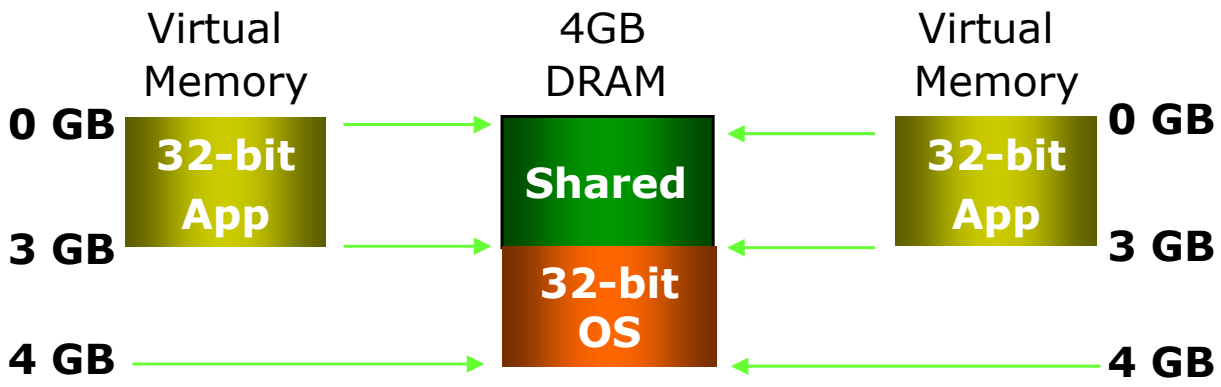
Growing Application Virtual Memory



Increased Memory for 32-bit Applications

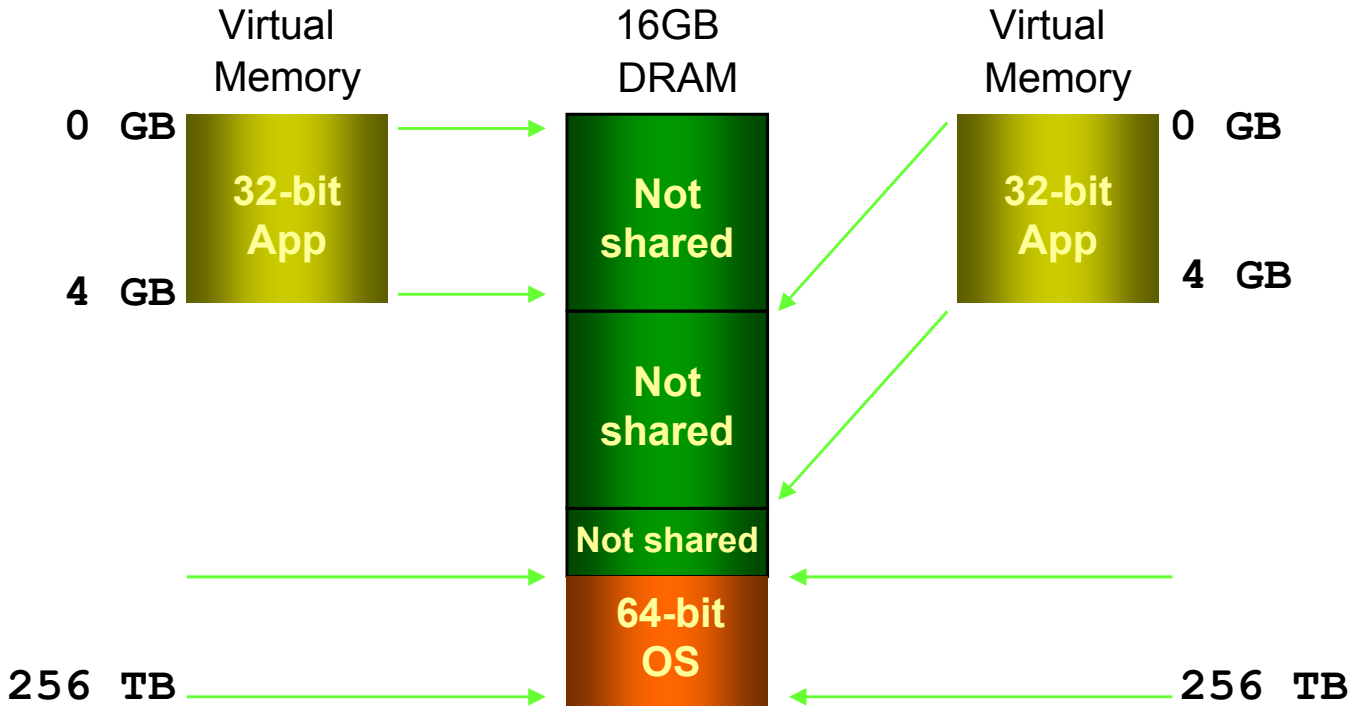
32-bit server, 4 GB DRAM

- OS & App share small 32-bit VM space
- 32-bit OS & applications all share 4GB DRAM
- Leads to small dataset sizes & lots of paging



64-bit server, 16 GB DRAM

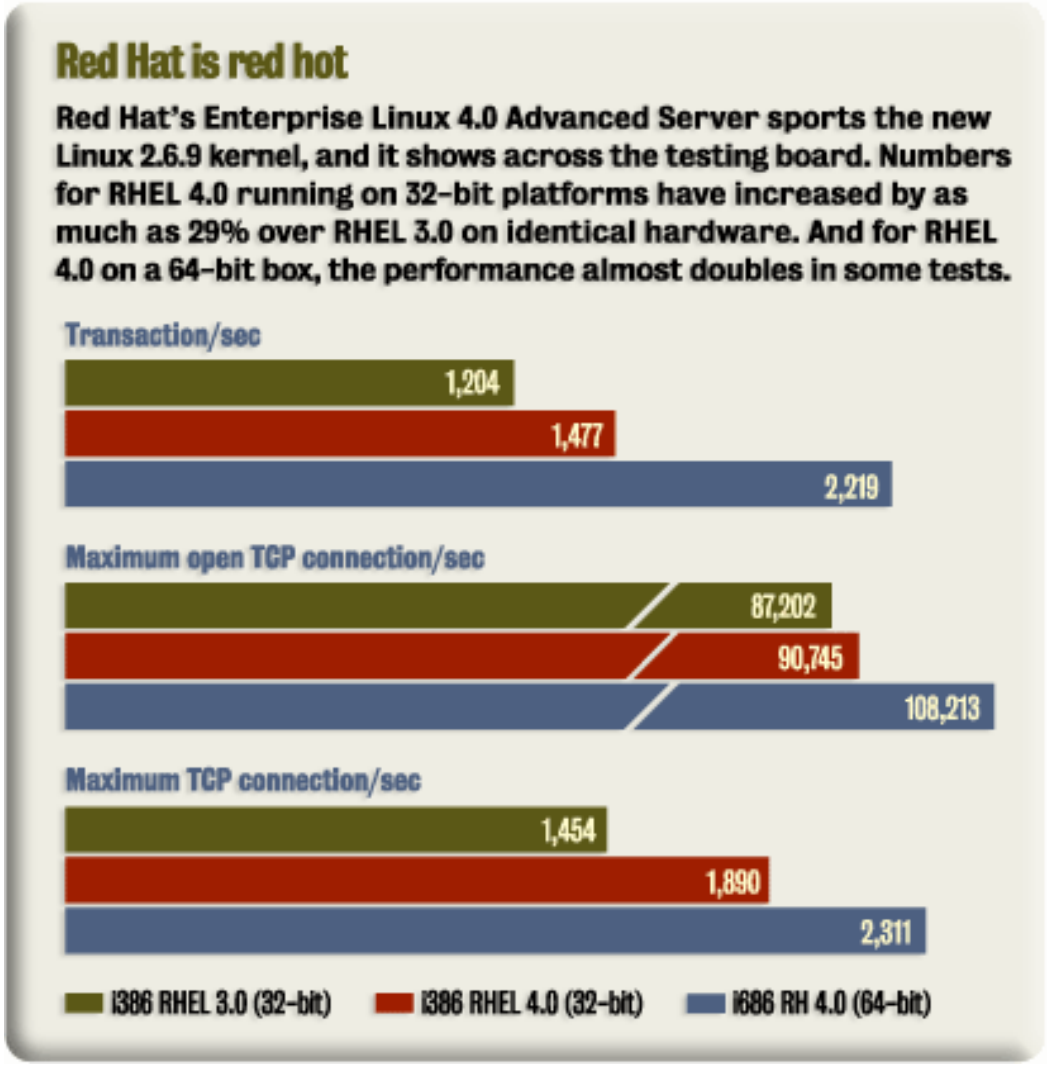
- App has exclusive use of 32-bit VM space
- 64-bit OS can allocate each application large dedicated portions of 16GB DRAM
- OS uses VM space way above 32-bits
- Leads to larger dataset sizes & reduced paging



Native Application FP on AMD64 Linux

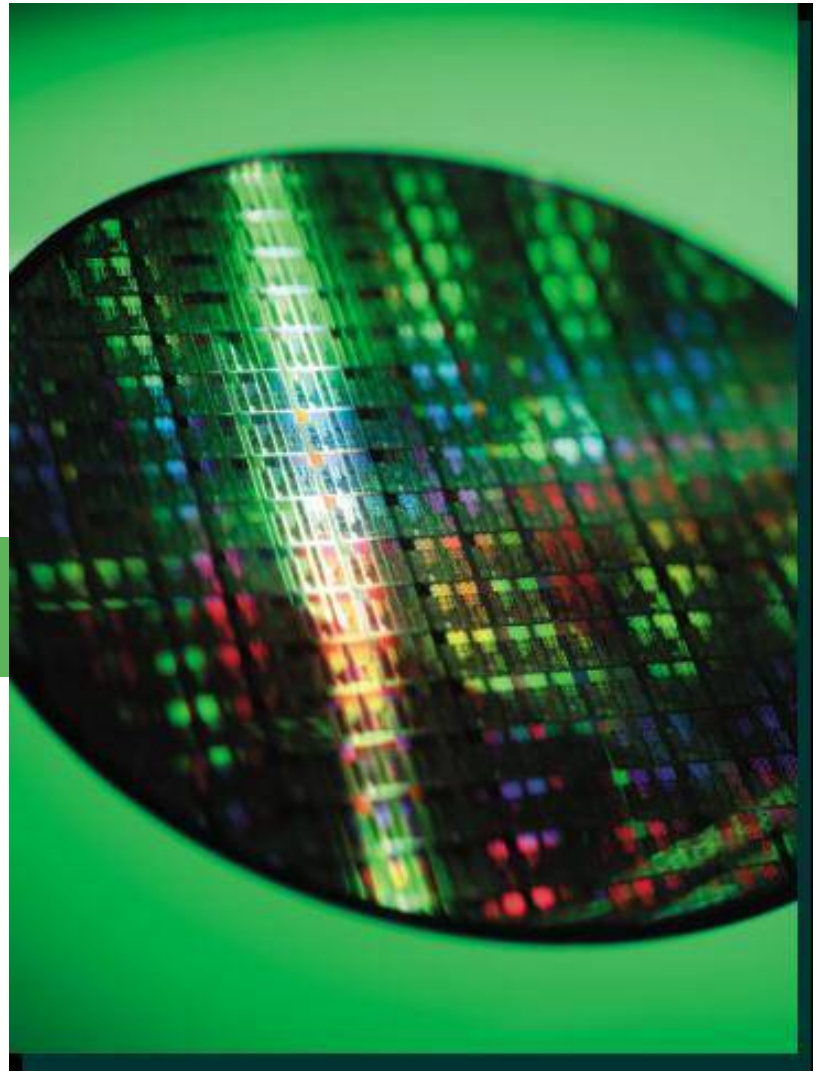
- Floating-point model uses SSE and SSE2 for highest performance and improved context switch time
  - 16 flat, 128-bit registers instead of 8-entry FP stack
  - Full Support for SSE/SSE2 compiler intrinsics
  - x87 is supported for 64-bit Applications thru "long double" data type.
  - All math routines in glibc will use just SSE/SSE2; all routines should match or exceed the performance of the legacy x87 routines

Red Hat Enterprise Linux 4.0
Congratulations!



In our Clear Choice test of this operating system package (we tested RHEL 4.0 Advanced Server, Red Hat's most robust Linux distribution), we found huge performance gains over previous editions, beefed up security options and vastly improved hardware detection mechanisms. For this combination, we give RHEL 4.0 a Network World Clear Choice award.

By Tom Henderson
Network World, 02/07/05

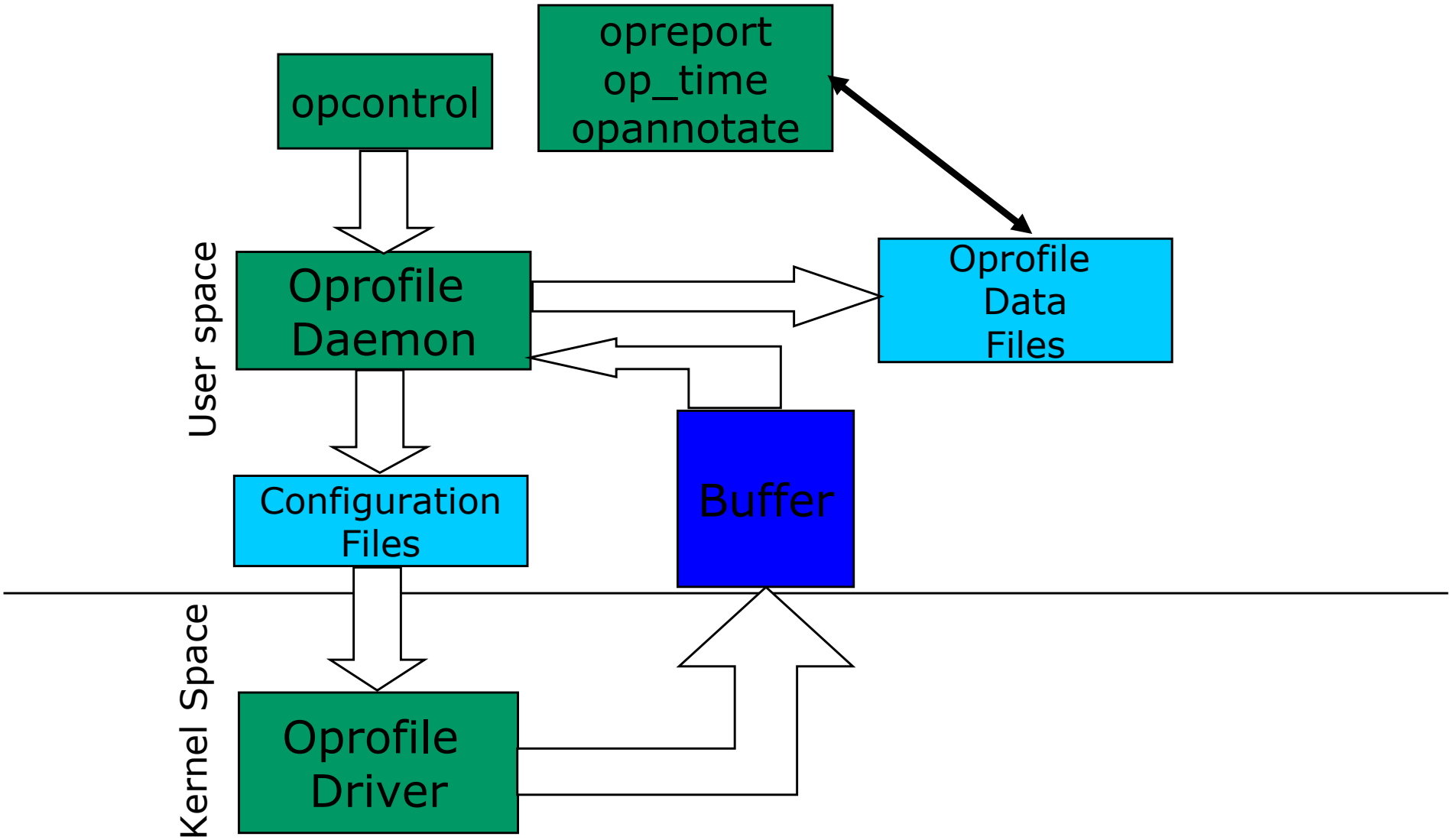


AMD64 Performance Optimizing

Oprofile Functionality

- Standard Linux tool: multi-platform wide support
- System wide profiling:
  - Oprofile is designed to profile the performance of binary modules, including user mode application modules and kernel mode driver modules.
- Dynamically loaded modules.
- Captures samples inside an interrupt service routine.
- Event-Based Profiling:
  - Oprofile event-based profiling (EBP) is designed to profile all 78 performance events of the AMD Opteron™ Processor and AMD Athlon™ 64 Processor, and event combinations.
  - Oprofile EBP is designed to profile up 4 events simultaneously. Event counts are displayed per CPU.
- Timer-based profiling provided by using '`CPU_CLK_UNHALTED`' event
- Multi-processor profiling: Oprofile can profile (both TBP and EBP) on multiple processor systems

Linux Oprofile Architecture



Oprofile commands

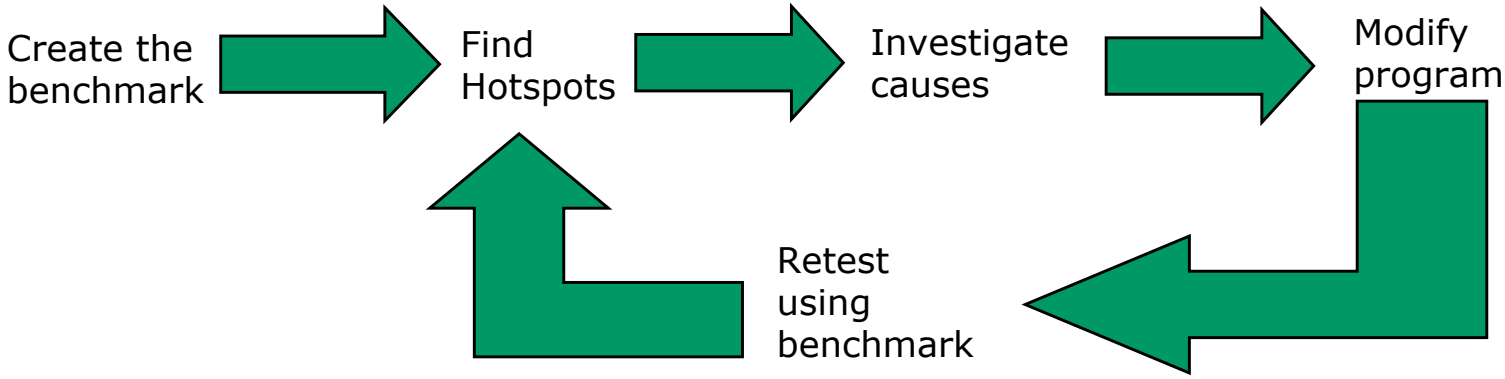
Oprofile 0.7.1

- `# opcontrol --list-events`
- `# opcontrol --setup --vmlinux=/boot/vmlinux \`
  `--event=CPU_CLK_UNHALTED:5000:0:1:1`
- `# opcontrol --start / --shutdown`
- `# opreport`
- `# opannotate`

Oprofile 0.5.4

- `# opcontrol -setup --no-vmlinux \`
  `-ctr0-event=CPU_CLK_UNHALTED \`
  `-ctr0-count=10000 -ctr0-unit-mask=0x1f`
- `# op_time, oprofpp, op_to_source`

Performance tuning process



The Software Optimization Process

- First step is to find hotspot. EBP's such as Oprofile are designed for identifying hotspots.
- Second step is to investigate hotspot to determine its cause.

- Inefficient algorithms
- Slow memory access
- High loop counts

- Branch prediction problem
- Slow instructions

What causes Hotspots and Cold-spots?

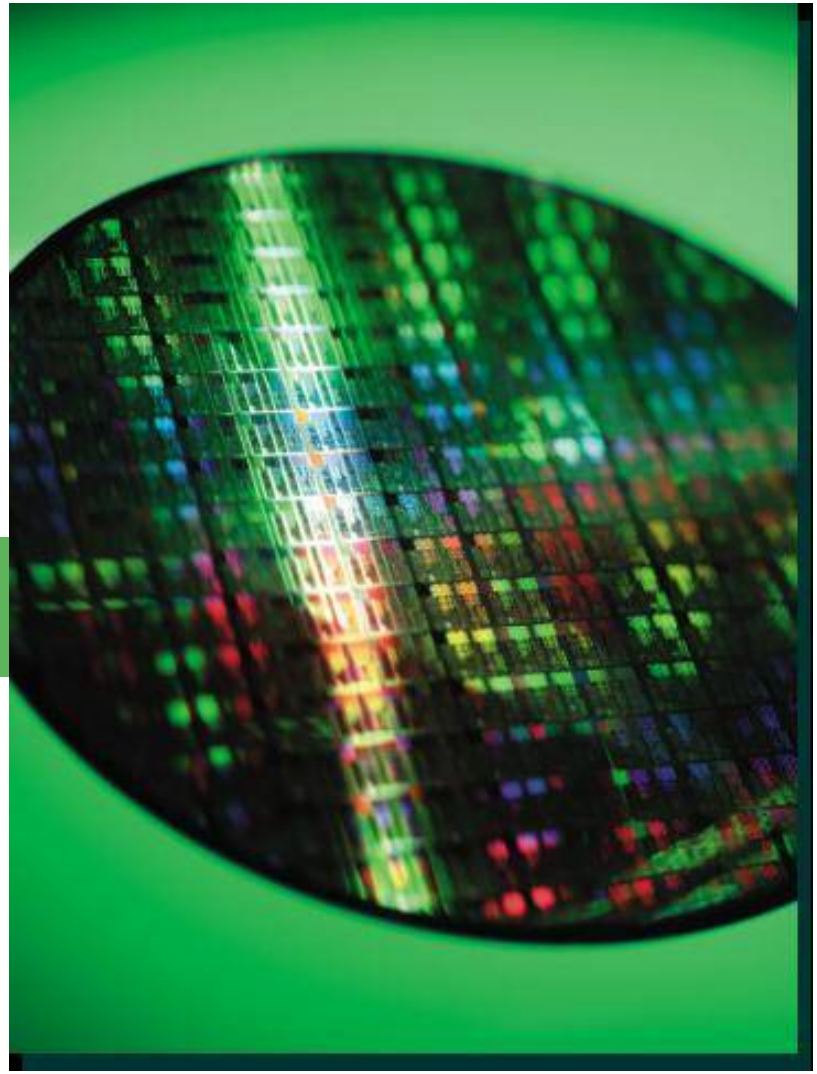
Application does not execute evenly. There are 3 major reasons that cause inconsistent execution:

1. Infrequent execution: initialization routine, error handling routine do not worth optimizing.
2. Slow execution: It could be hotspot only if it consumes significant time compared to the rest of application.
3. Frequent execution: hotspot if it consumes significant time.

- Hotspots are the areas of the application that have intense activity. It's the place to start optimizing.
- Intense activity usually refers to time, but the reasons that causes hotspot could be: braches mis-prediction, cache misses etc.

AMD CodeAnalyst

- AMD designed and developed profiling tool
  - Advanced GUI based
    - *Qt interface*
    - *vs. Oprofile's comand-line approach*
  - Distributed as open source tarball, or pre-compiled RPM's.
  - Provides Event-based Profiling & Timer-based profiling.
  - Covered in depth in Sherry Hurwitz's presentation, previous to this presentation.



Linux x86-64 Gotchas

Correct return type declarations for ptrs

- Functions returning pointers (& longs) must be prototyped to return a pointer (long).
- When the return value is not explicitly declared, the function is assumed to return a 32bit int.
  - Which leads to a truncation of the pointer and a crash on accessing the pointer.
  - Common victims of this are standard library functions like strerror or malloc when their include file is not included.
- How to check:
  - Make sure your prototypes are correct and consistent. Look for prototypes that are not declared in shared include files and verify that they are correct.
  - Run LClint with only function call checking enabled over the whole program to check for consistent prototypes.
  - The gcc protoize tool may also be used to generate a global prototype file that could be included in every file using gcc's `-include` option.
  - You should also compile with the gcc "`-Wall`" or "`-Wmissing-prototypes`" switches and check for the warnings. Also, look for warnings about undeclared functions.

Correct Prototypes when using stdargs

- Some programs have inconsistent prototypes over multiple source code files.
- By convention, stdargs and varargs use the %al register to indicate the number of floating-point arguments.
- The entry point of a function using stdargs or varargs, expects the %al register to be initialized properly.
- If the function prototype used by caller doesn't include the "(...)", gcc won't initialize %al as part of the call.
  - And a crash and/or strange behavior will occur after the call.
- glibc defines some functions unexpectedly as stdarg (e.g. ptrace, fcntl, open) and expects %al to be correct.
  - So it is essential that these have correct prototypes.
  - Best done by including the correct glibc header.

Adhere to ISO C specification for stdargs

- Do not copy ‘ap’ directly:

```
void foo (va_list ap) {
        va_list tap = ap;
        /* use ‘tap’ */
}
```

- Correct usage:

```
#include <stdargs.h>
void foo (va_list ap) {
        va_list tap;
        va_copy (tap, ap);          /* required for same reason as strcpy(3) */
        /* use ‘tap’ */
}
```

- GCC’s ‘ap’ is a pointer for AMD64.
  ‘&ap’ does not do what you may expect.
- Review ISO C99 standard, section 7.15.

64-bit Library Path

- 64-bit libraries are in */lib64 instead of */lib.
  - 32-bit libraries remain in */lib.
- Makefiles that install libraries must be changed for this.
- Also the common -L/usr/X11R6/lib compile flag or a similar flag for the Qt library must be changed to reference */lib64.
- GNU autoconf will handle this with
  `"configure --enable-lib-suffix=64"`
- Consider using GNU libtool.
- gcc 3.3 conversion bump …, -Wall gives much more warnings, much, much more picky.

Glibc Malloc(3)

- Sometimes uses sbrk(2)
  Sometimes uses mmap(2)
- Can return wildly different address for allocated memory.
- Use mallopt() to control behavior. Especially when linking mix-mode code (especially FORTRAN).
- Use ‘ptrdiff_t’ (signed 64-bits) or `size_t’ (unsigned 64-bits) to index huge arrays, not ‘int’ (signed 32-bits).

X86-64 Backtrace and Stack Unwinding

Differences from 32-bit x86

- Parameters passed in registers vs. on the stack.
  - Parameters can be "lost" if register reused for another purpose and value not saved.
  - Application-specific stack unwinders are more complicated to impliment and not a simple port from the 32-bit implimentation. Glibc provides helper function.
- Default is `-fno-emit-framepointer`
  - Frees up rbp for other uses, but makes it harder to unwind the call stack without "`-g`".
- Stack alignment maintained at 64-bits
- ABI specifies tail-call elimination
  - Leaf functions can "disappear"
- DWARF2 debugging information.
  - Used for both debugging information and OOP exception handling.
  - Compiling "`-g`" can cause binaries to be much larger.

Memory Model Facts

- IP relative addressing is cheaper than direct-addressing
- Immediates/displacements remains 32-bit sign extended
- 32-bit operations zero extend
- movabs instruction to load 64-bit immediates available

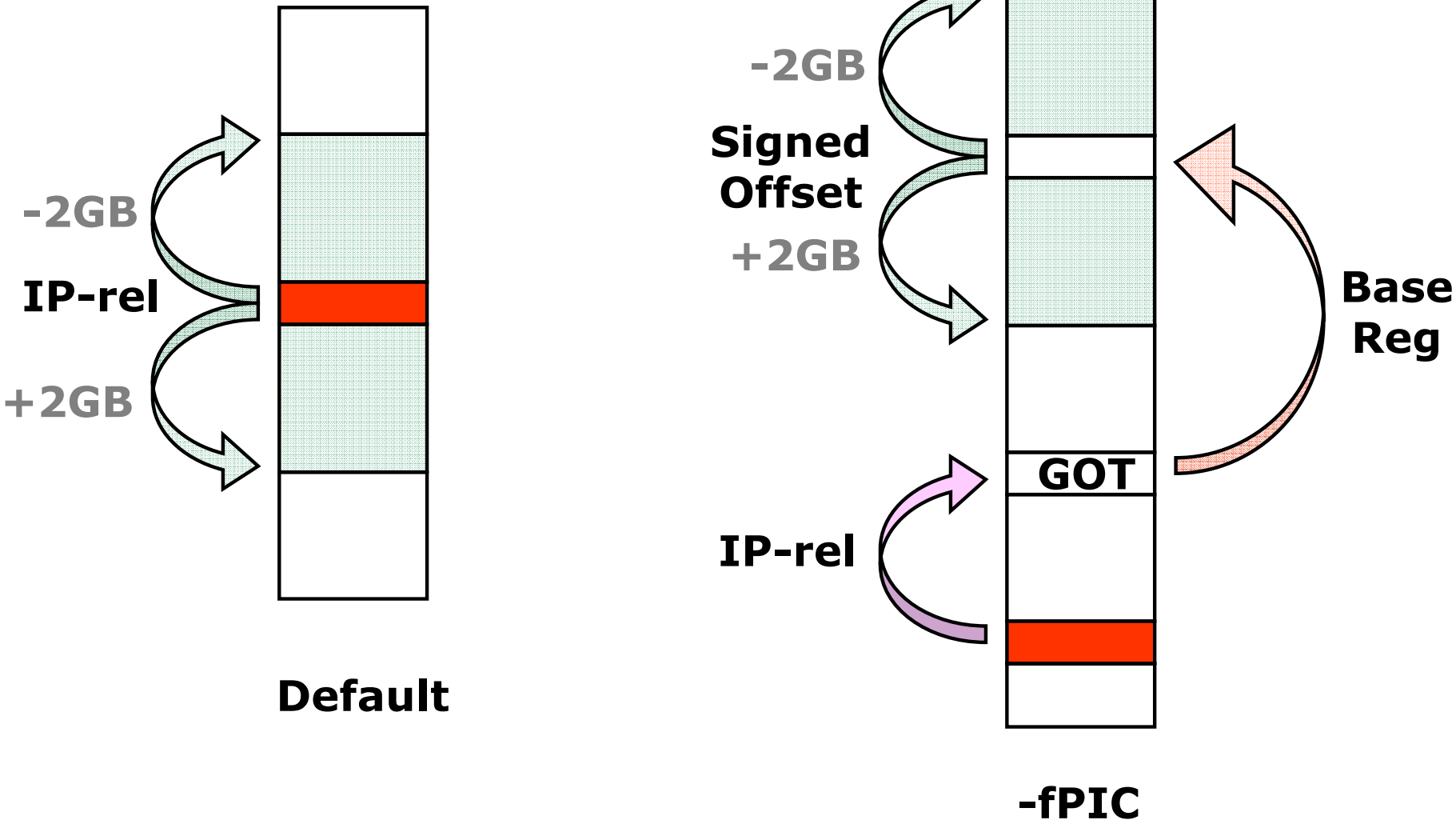
Shared libs must be compiled with -fPIC

- Don't rely on 32 bit relocations sometimes working between shared libraries when they are loaded at startup.
  - Dumb Luck: because the shared libraries are within reach of 32 bit offset
  - but it may fail for shared libraries that are loaded later with dlopen() for example.
- This problem only affects 64-bit libraries. Libraries compiled for 32-bit programs may continue to be compiled w/o `-fPIC`.
- How to check:
  - The linker (ld) will complain if it encounters a library not compiled with `-fPIC`. It will also fail to link against the library.
  - If a library fails to link, you can check it by running `readelf -r` and `nm` on it.
  - Relocations shown as "U" in nm must have a corresponding entry in the GOT table under '`.rela.got`'.

Shared libs must be compiled with -fPIC

- i386 and a few other ports tolerate shared libraries that are not compiled with `-fPIC`.
-
- Compiling 64-bit AMD64 libraries without `-fPIC` leads to crashes when accessing external symbols to the shared library.
  - e.g. symbols declared in the main program.
  - This is because only 32-bit relocations are used for referencing the symbols
  - But the shared library is loaded more than 32 bits away from the main program.

Shared Lib Access to Global Symbol



Small Model *(default)*, Absolute

```
extern int src;
extern int dst;
extern int *ptr;

dst = src;


ptr = &dst;

*ptr = src;
```

```
.extern src
.extern dst
.extern ptr
.text

movl src(%rip), %eax
movl %eax, dst(%rip)

lea dst(%rip),%rdx
movq %rdx, ptr(%rip)

movq ptr(%rip),%rax
movl src(%rip),%edx
movl %edx, (%rax)
```

- Size of code + data is 2GB
- Virtual address of code executed is known at link time.
- All Symbols are known at link time.
- All symbols in range from 0 to 2^{31}-2^{10-1} (2GB)
- %rip-relative addressing can be used
- Provides Fastest Performance

Medium Model, Absolute

```
extern int src;
extern int dst;
extern int *ptr;


dst = src;

ptr = &dst;


*ptr = src;
```

```
.extern src
.extern dst
.extern ptr
.text

movabsl   src, %eax
movabsl   %eax, dst

movabsq  $dst,%rdx
movabsq  %rdx, ptr

movabsq  ptr,%rdx
movabsl   src,%eax
movl      %eax, (%rdx)
```

- Size of text section is 2GB
- Virtual address of code executed is known at link time.
- All Symbols are known at link time.
- Can't assume range of symbolic references to data sections.
- Must use movabs to access static data and load addresses into register

Position-Independent-Code (PIC) Models

- GOT, PLT and static data accessed by %rip-relative addressing

Small PIC

- Size of code + data is 2GB
- Virtual address of code executed **not** known at link time.
- Symbols **not** known at link time.
- All symbols and code within 2GB of where code is loaded
- %rip-relative addressing can be used for both code and data

Medium PIC

- Size of code is 2GB, %rip-relative can be used for code
- Virtual address of code executed **not** known at link time.
- Symbols **not** known at link time.
- All code within 2GB of where code is loaded
- Can't assume range of symbolic references to data sections.
- %rip-relative addressing can't reach all static data
- Must use movabs to access static data and load addresses into register

PIC Example

```
extern int src;
extern int dst;
extern int *ptr;
```

```
.extern src
.extern dst
.extern ptr
.text
```

```
dst = src;
```

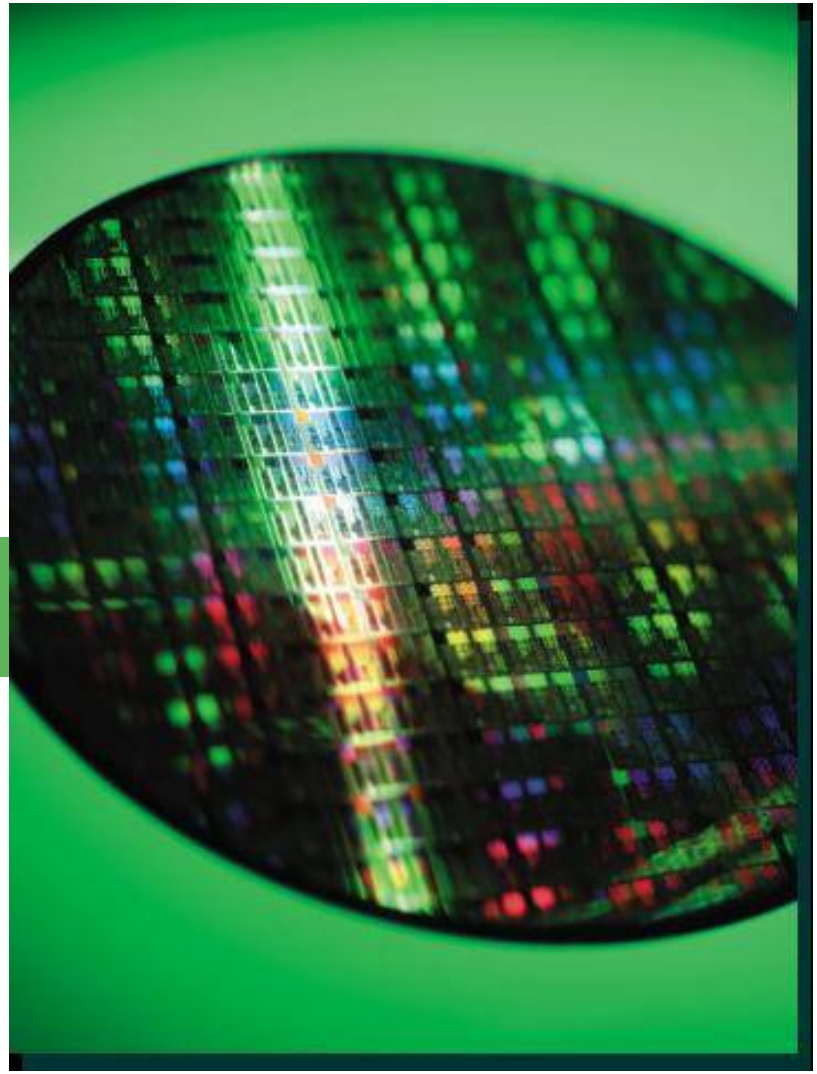
```
movq src@GOTPCREL(%rip), %rax
movl (%rax), %edx
movq dst@GOTPCREL(%rip), %rax
movl %edx, (%rax)
```

```
ptr = &dst;
```

```
movq ptr@GOTPCREL(%rip), %rax
movq dst@GOTPCREL(%rip), %rdx
movq %rdx, (%rax)
```

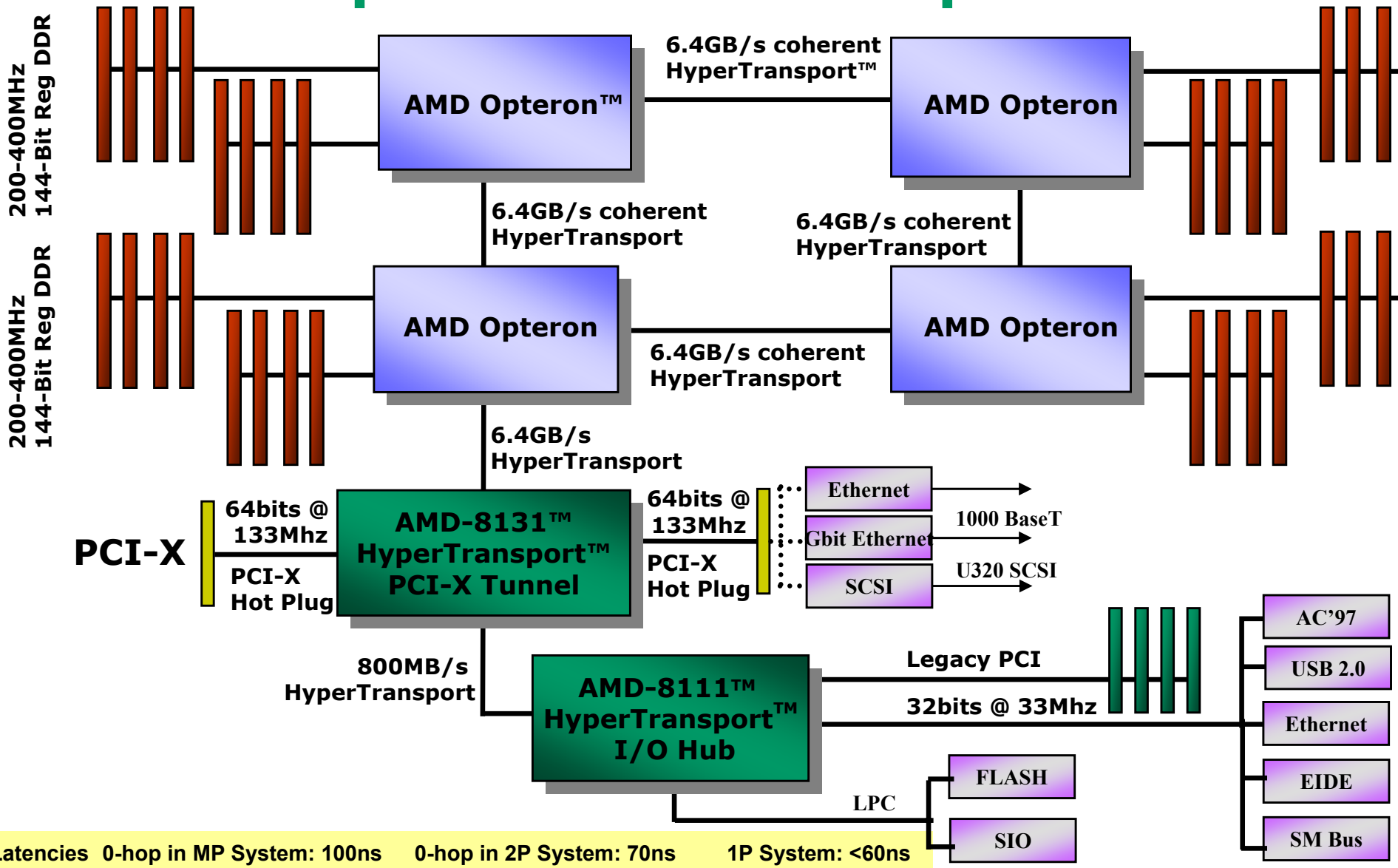
```
*ptr = src;
```

```
movq ptr@GOTPCREL(%rip),%rax
movq (%rax), %rdx
movq src@GOTPCREL(%rip), %rax
movl (%rax), %eax
movl %eax, (%rdx)
```



Memory & Multi-Processor Performance on AMD64

4-Socket Opteron™ Server Implementation



Idle Latencies to First Data:	0-hop in MP System: 100ns 1-hop in MP System: 110ns 2-hop in MP System: 135ns 3-hop in MP System: <180ns	0-hop in 2P System: 70ns 1-hop in 2P System: 105ns	1P System: <60ns

NUMA Terms

Non-Uniform-Memory-Architecture

- "node": A grouping of processors & memory modules where:
  - all processors within the group have roughly identical memory access time and latency to memory within the group.
  - all processors within the group have roughly identical memory access time and latency to memory outside the group.
  - Caches are ignored for this definition.

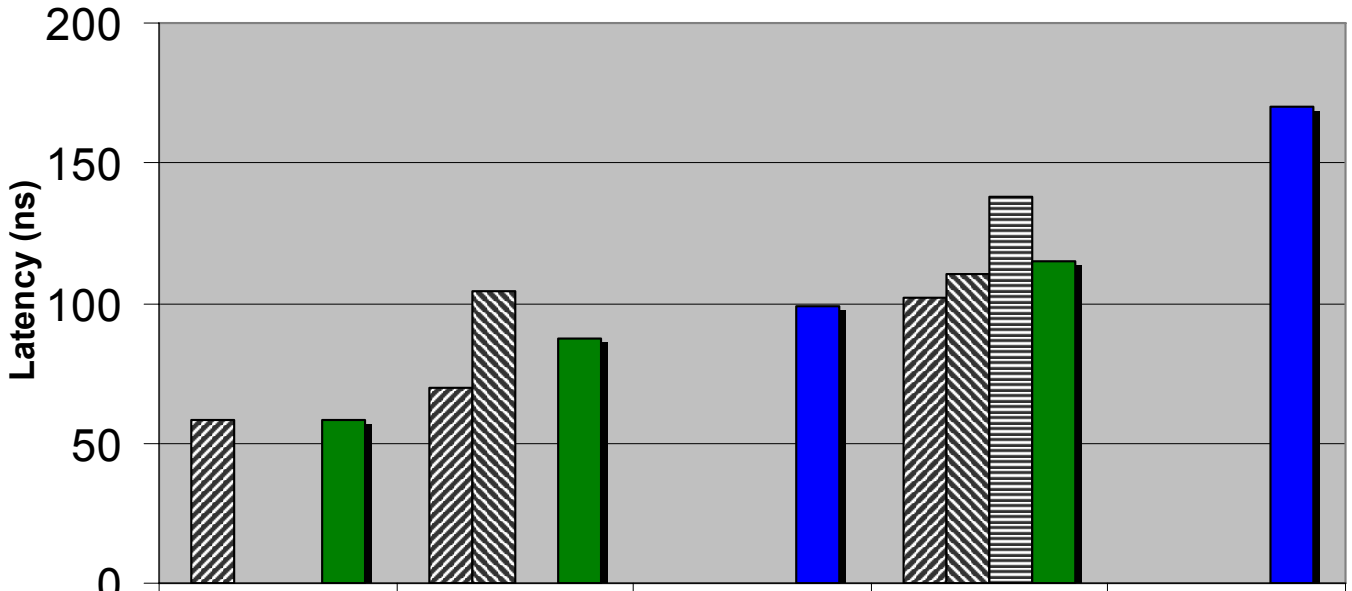
- "NUMA Factor": calculated as the ratio of the average remote time to the average local time for a given workload.
  - AMD Opteron™ well below the "2:1" rule of thumb.
  - 2-hop access latency of 135ns vs 0-hop latency of 100ns on 4P machine implies a 'NUMA Factor' of 1.35.
  - 1-hop access latency of 105ns vs. 0-hop latency of 70ns on 2P => 1.5
  - Empirically measured to be 1.3 on server benchmark workloads; thus very low on a 4P AMD Opteron™ system across a broad range of workloads.

Simple Software Model for NUMA

- NUMA bring dramatic scalability advantages
  - But software management is hard to get right
- SMP systems bring dramatically simplified software model
  - But memory system doesn't scale up as you add processors
- AMD Opteron™ processor provides benefits of both: "SUMA" Sufficiently Uniform Memory Architecture, gives SMP view for software
  - Physical address space is flat and fully coherent
  - Latency difference between local and remote memory in an 8P system is comparable to the difference between a DRAM page hit and page miss
  - DRAM can be contiguous or interleaved
- "ccNUMA" Cache-coherent Non-Uniform Memory Architecture
  - Latency shrinks quickly by increasing CPU and HyperTransport ™ technology link speed
  - Additional processor nodes bring increased memory bandwidth and great overall system throughput

Memory Latency

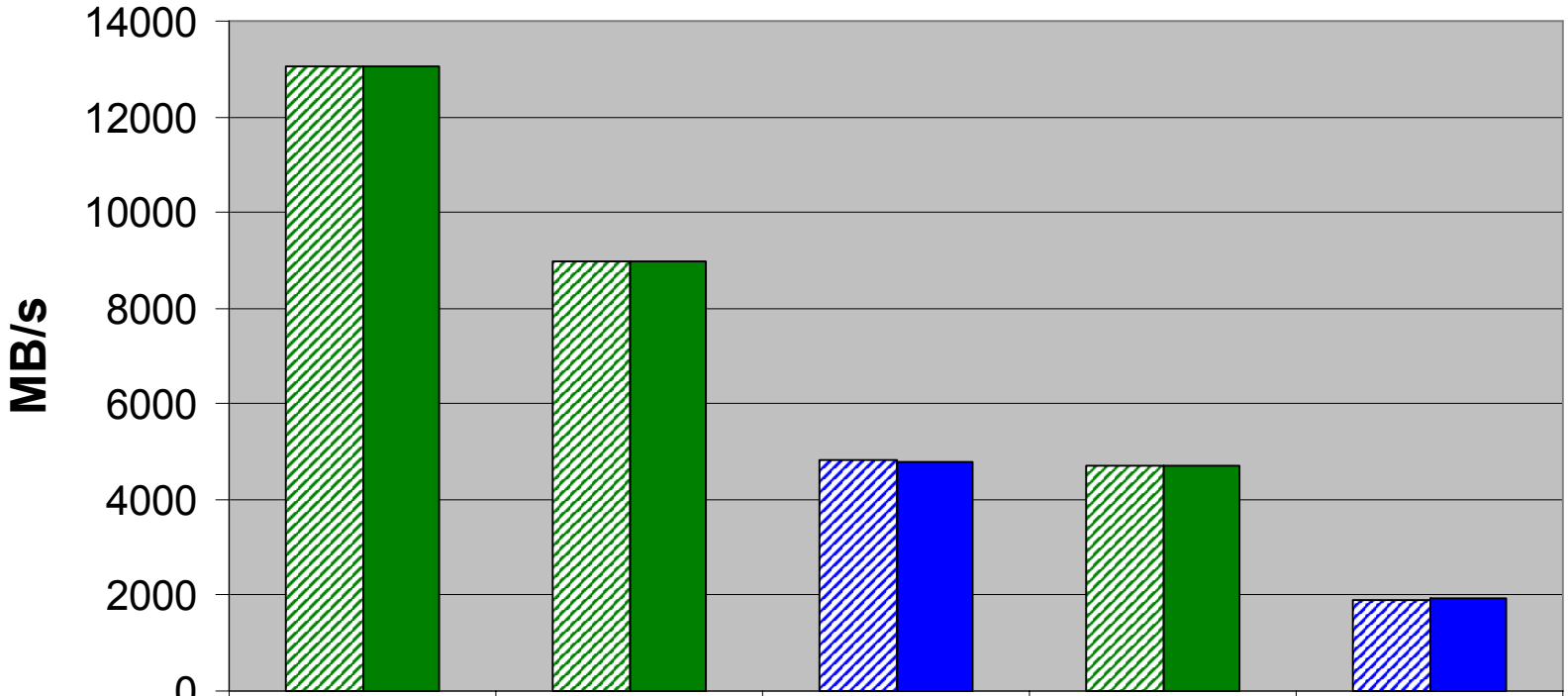
ScienceMark 2.0 Beta,
512-Byte Stride Latency (ns)



	1 AMD Opteron™ Model 146 2xPC2700 CL2.5	2 AMD Opteron Model 246 4xPC2700 CL2.5	1 P4 3.0GHz, 800 FSB, 875, 4xPC3200 CL2	4 AMD Opteron Model 846 8xPC2700 CL2.5	Dell PE 6600 4 XeonMP 2GHz 16xPC1600 CL2
0-hop (512 byte stride, ns)	58	70		102	
1-hop (512 byte stride, ns)		104		110	
2-hop (512 byte stride, ns)				138	
Weighted Average	58	87	99	115	170

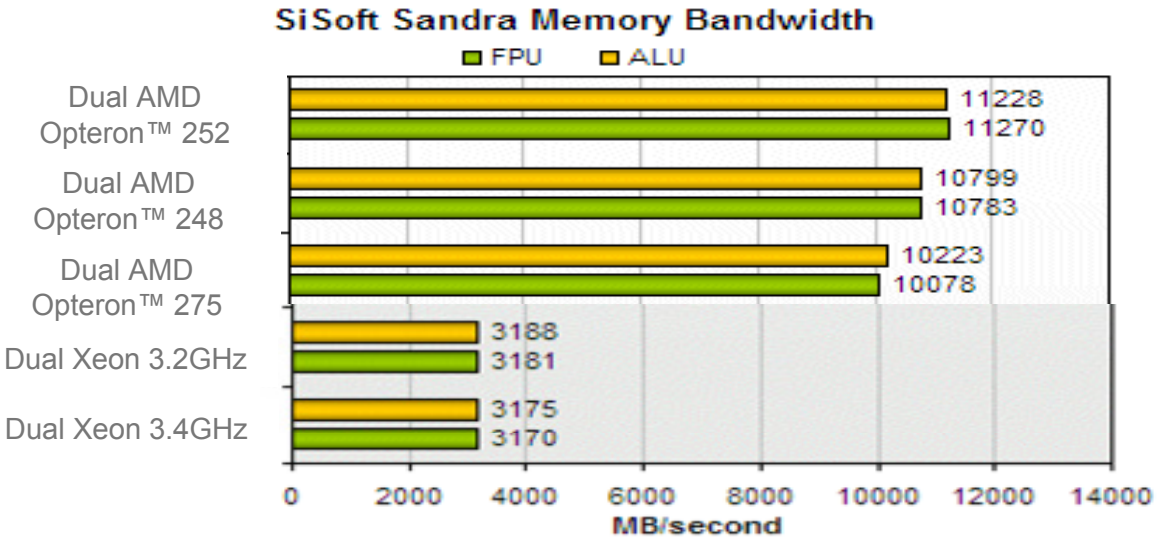
Memory Bandwidth

Sisoft Sandra Standard 2003/SP1 9.44



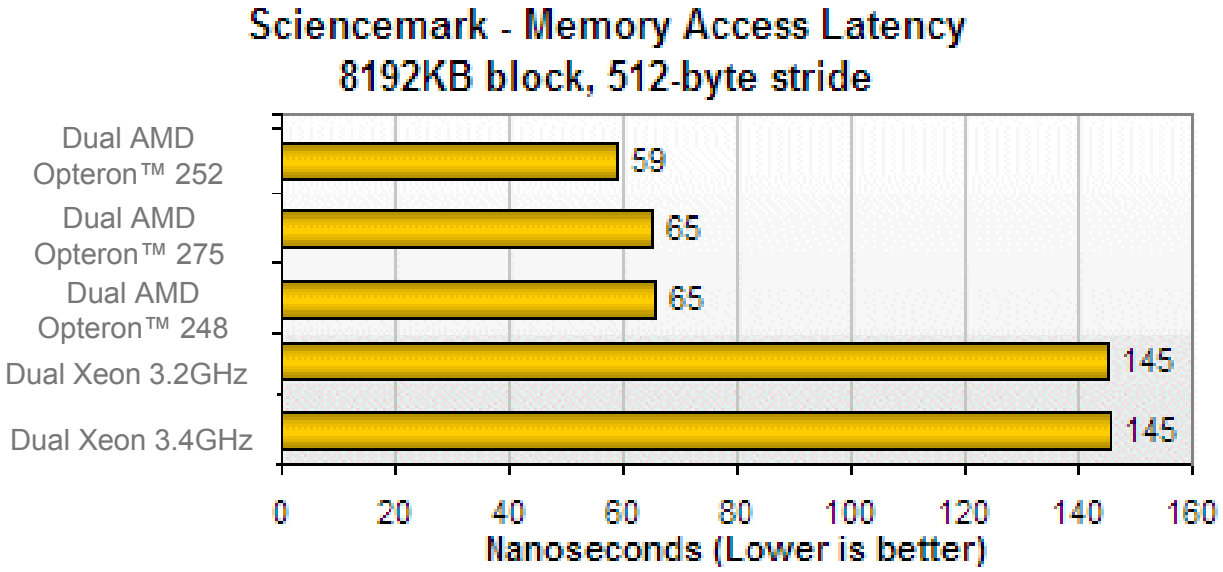
	4 AMD Opteron™ Model 846 8xPC2700 CL2.5	2 AMD Opteron Model 246 4xPC2700 CL2.5	1 P4 3.0GHz 800FSB, 875, 4xPC3200 CL2	1 AMD Opteron Model 146 2xPC2700 CL2.5	4 Xeon MP 2GHz, GC-HE, 16xPC1600 CL2
RAM Bandwidth Int Buff iSSE2 (MB/s)	13060	8970	4808	4707	1912
RAM Bandwidth Fp Buff iSSE2 (MB/s)	13057	8965	4797	4720	1920

Results: Fast Access to Memory



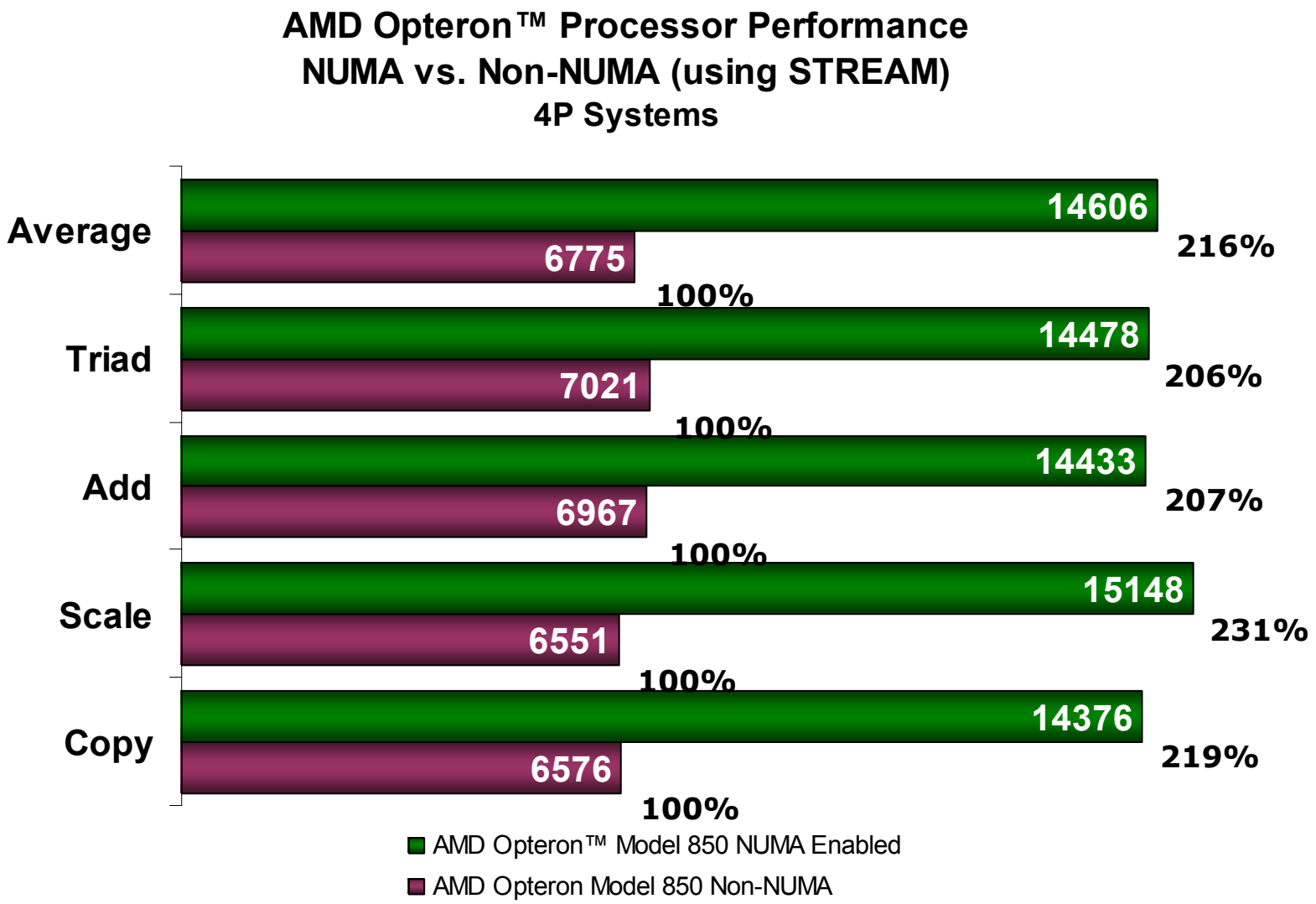
Memory bandwidth increases with frequency

Memory latency decreases with frequency



http://techreport.com/reviews/2005q2/opteron-x75/index.x?pg=5

NUMA vs. Non-NUMA (using STREAM)



64-bit Linux kernels that support Opteron NUMA

- Will be labeled as "SMP" kernels, even though they support NUMA

- All 2.6 based kernels. Examples:

  SuSE Linux Enterprise Server 9, 2.6.5

  SuSE Linux Pro 9.x, 2.6.5 & SuSE Linux Pro 10.x

  Red Hat Enterprise Linux 9, 2.6.5

  Fedora Core 2 and later

- Some 2.4 based kernels, including:

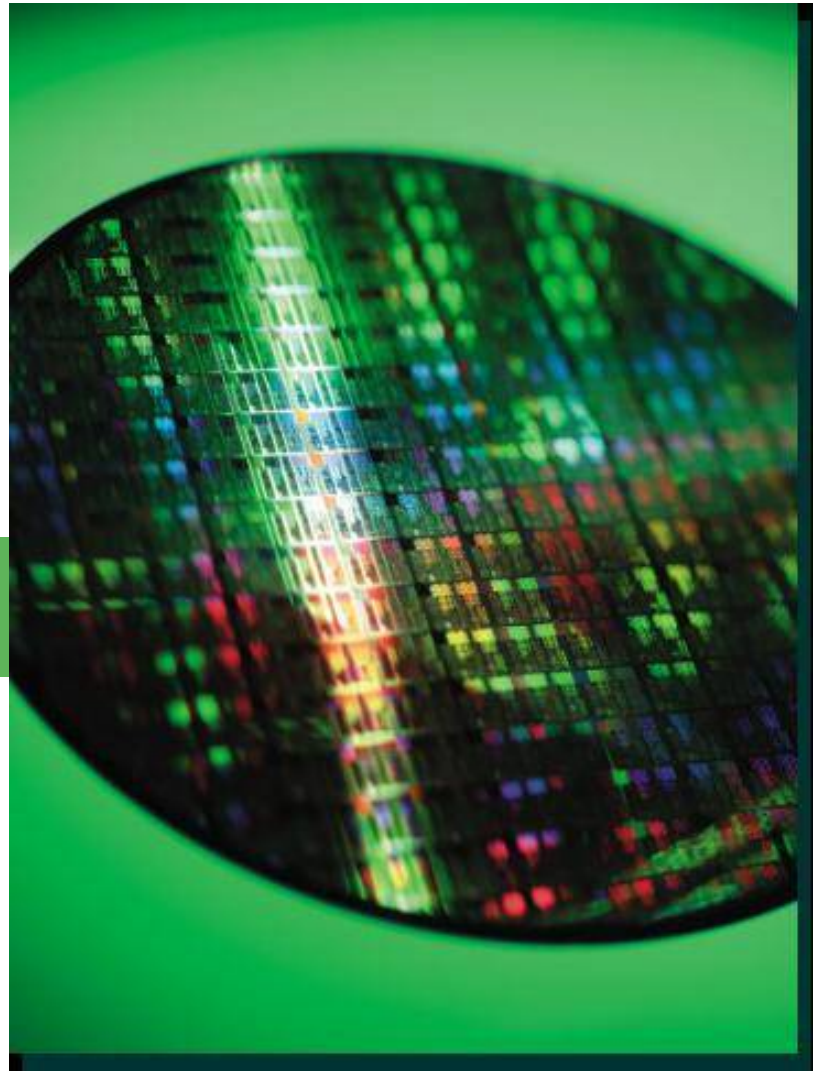
  SuSE

  SuSE Linux Enterprise Server 8 SP2, 2.4.19-249

  SuSE Linux Enterprise Server 8 SP3, 2.4.21-143

  Redhat

  Redhat Enterprise Linux 3.0 update 2, 2.4.21-15

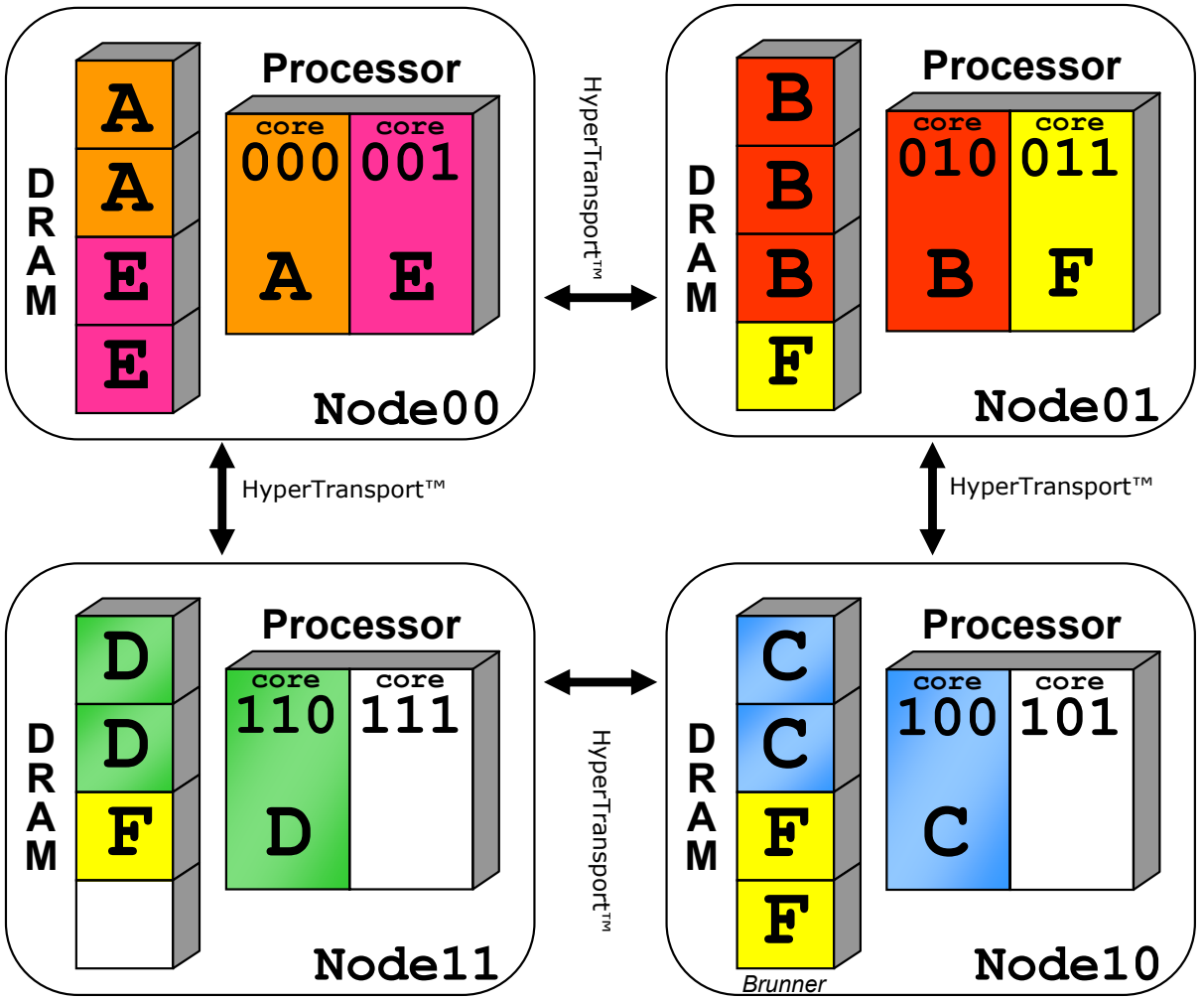


NUMA Support in the Linux Kernel for AMD64

NUMA OS Dual-Core Scheduling Example

Example of an OS optimized for dual core running on an AMD Opteron™ system with 4 dual core processors

- OS distributes threads across processors *first* to minimize contention between threads
- OS attempts to map a thread's memory requests to the local memory of the node (Threads A, B, C, D, E)
- If no memory is left on the current node, OS will use memory of other nodes (Thread F)



Run in slide show mode to view animation of OS scheduling for dual core

Homenode Scheduling (1)

- Based on O(1) scheduler
- "Homenode": an App is assigned a "Home" at exec time:
  - The App's memory is preferably allocated at this Home node.
  - The App is scheduled to this Home node unless it misbalances the machine too much.
  - It's a tradeoff between CPU usage and memory latency.
- Adds statistics for better counters (these are exact) of node loads.
- Support for interleaving of large page shared memory (with other patches).
  - This is mainly for databases.

Homenode Scheduling (2)

- Basic idea: trade some SMP tuning (well balanced scheduling, aggressive CPU affinity) for more overall memory bandwidth and less latency for local memory.
  - Pro: machine performance is constant on average (less performance fluctuations due to cross-node traffic, predictable)
  - Con: when the algorithms get it wrong you get a slow down.
- Various policies to assign homenodes initially to processes:
  - assign at exec time only
  - assign at exec fork
  - assign at exec and clone (= for threads), but not fork; this is the default
- Process is migrated to new node. This makes the CPU affinity of child processes weaker, but helps longer term.
- Does migration on wakeup unlike normal O(1) scheduler
- When the machine is very unbalanced the process will not be scheduled on the homenode.

Description of NUMACTL

- High level command line control for NUMA (use like `nice(1)`)
  - Changes policy process and all its children.
  - For fine-grained granularity (per thread etc.) use libnuma instead
  - Currently "node" corresponds to CPU number on AMD64.
  - Once a fixed policy is set it is used by all children, the automatic load balancing is turned off.
- numactl --interleave=nodelist child ...
  - Interleave memory in child. May not be a good idea for all memory, but is useful for testing.
- numactl --membind=nodelist --cpubind=nodelist child ...
  - Bind CPU and memory to a specific node
  - may be useful to combine with irq affinity using /proc/irq/*/irq_affinity
- numactl --localalloc child ...
  - Force local alloc policy. Only useful when the global system policy is different
- numactl --homenode=node child...
  - Set homenode to node. This is like cpubind/membind, just a bit weaker and can be overriden by the system if needed.
- numactl -s
  - Show current NUMA state.
- numactl –hardware
  - Print an overview of the available nodes.

Controlling CPU Core / Memory Affinity

"numactl" command allows control of core/memory affinity for a process "test.exe":

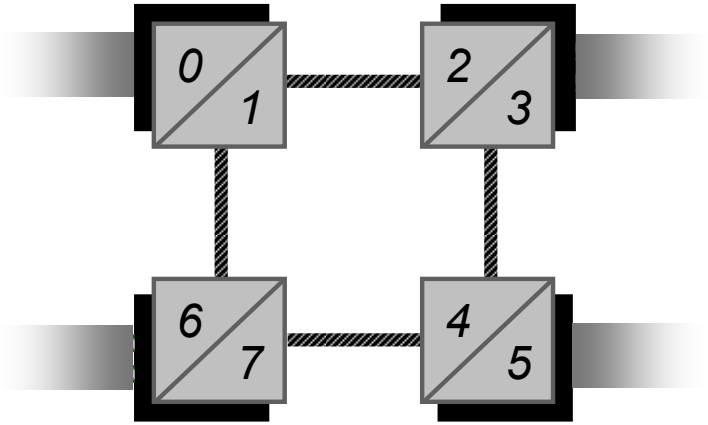
EXAMPLE: system with 4 Dual-Core Opteron's (8 cores)

- run upon 4 sockets using only 1 of every 2 cores

```
$ numactl –cpubind=0,2,4,8 –membind=0,2,4,8
test.exe
```

- run upon 4 sockets but interleave memory for process across all 4 IO controllers

```
$ numactl –interleave=0,2,4,8 test.exe
```



libnuma on AMD64 (1)

- libnuma offers a simple programming interface to the NUMA policy supported by the Linux kernel.
  - "Builds NUMA support into the application" vs. using numactl.
- For setting global policy per process use the numactl utility.
- Available policies are page interleaving, home node allocation, local allocation, allocation only on specific nodes.
- A policy exists per thread, but is inherited to children.
- All NUMA memory allocation policies only takes effect when a page is actually faulted into the address space of a process by accessing it (e.g. "lazy allocation").
  - The NUMA_alloc_* functions take care of this automatically.
- Some of the NUMA functions use a nodemask. A nodemask is a bitmask where each bit corresponds to a node.
  - Bit 0 refers to node 0 which contains the Boot-Services-Processor.
  - A nodemask should be only manipulated with specific nodemask functions.

Key libnuma functions: Processor Affinity

- numa_set_homenode & numa_homenode
  - sets (or returns) the homenode for the current thread.
  - the homenode stays locked for the current thread and its children until it is reset using numa_set_homenode(-1).
- numa_bind
  - binds the current thread and its children to the nodes specified in nodemask.
  - They will only run on the CPUs of the specified nodes and only able to allocate memory from them.
  - This function is equivalent to calling numa_run_on_node_mask and numa_set_membind with the same argument.
- numa_run_on_node & numa_run_on_node_mask
  - runs the current thread and its children on a specific node (or on nodes in nodemask).
  - They will not migrate to CPUs of other nodes until the node affinity is reset with a new call numa_run_on_node_mask.

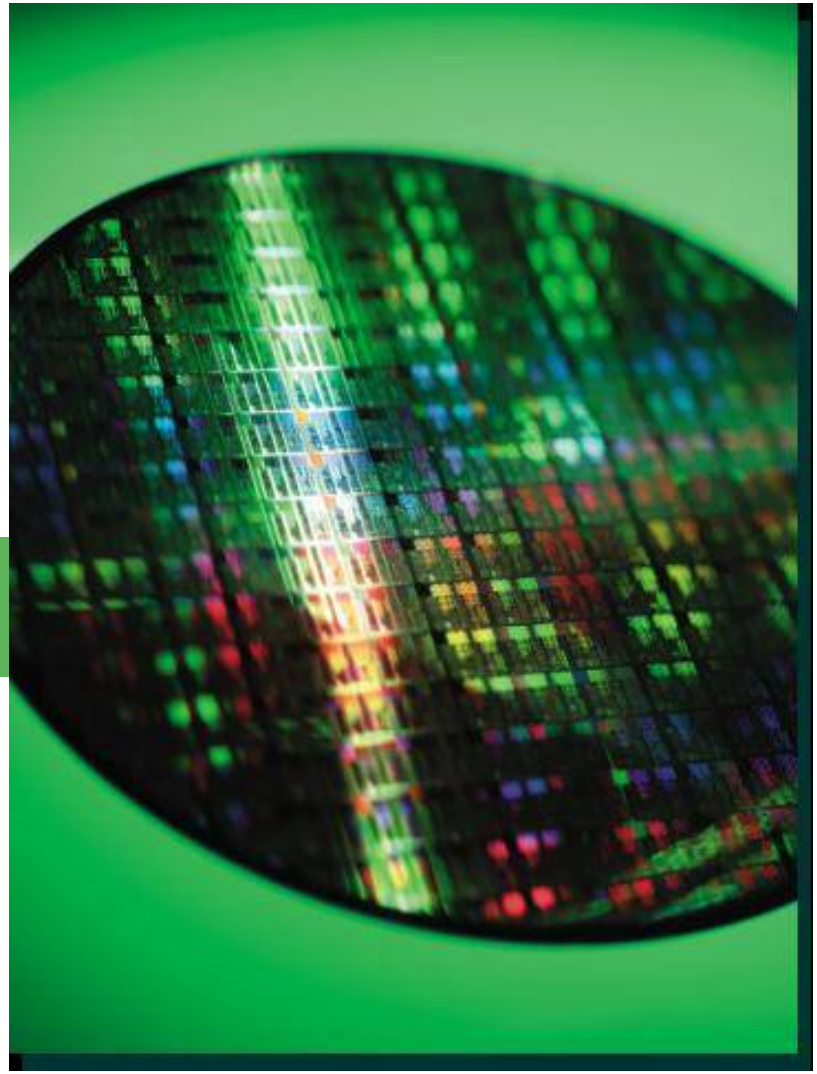
Key libnuma functions: Memory Affinity

- numa_set_localalloc
  - sets a local memory allocation policy for the current thread based on a flag.
  - Non-null Flag: attempt to allocate memory from current node.
  - Null Flag: attempt to allocate memory from homenode.
- numa_set_membind & numa_get_membind
  - Sets/Gets the memory allocation mask. Thread will only allocate memory from nodes set in nodemask.
  - Setting argument of numa_no_nodes or numa_all_nodes turns off memory binding to specific nodes.
  - Getting return value of numa_no_nodes or numa_all_nodes means all nodes are available for memory allocation.
  - Currently, VM is unaffected by this policy, so swapping is not heavily influenced by this policy. (Future work planned).

Key libnuma functions: Memory Alloc Policy

- numa_free
  - Only correct way to free memory allocated by the numa_alloc_* functions.
- numa_alloc_interleaved & numa_alloc_interleaved_subset
  - allocates memory as page interleaved on all nodes (or subset specified by nodemask).
  - interleaving works on page level and will only show an effect when the area is large.
- numa_alloc_onnode & numa_alloc_local
  - allocates memory on a specific node (or local node).
  - allocations are rounded to pagesize.
- numa_alloc
  - allocates memory with the current NUMA policy.
  - allocations are rounded to pagesize.

Trademark Attribution



32-bit Apps on a 64-bit Linux running on AMD64

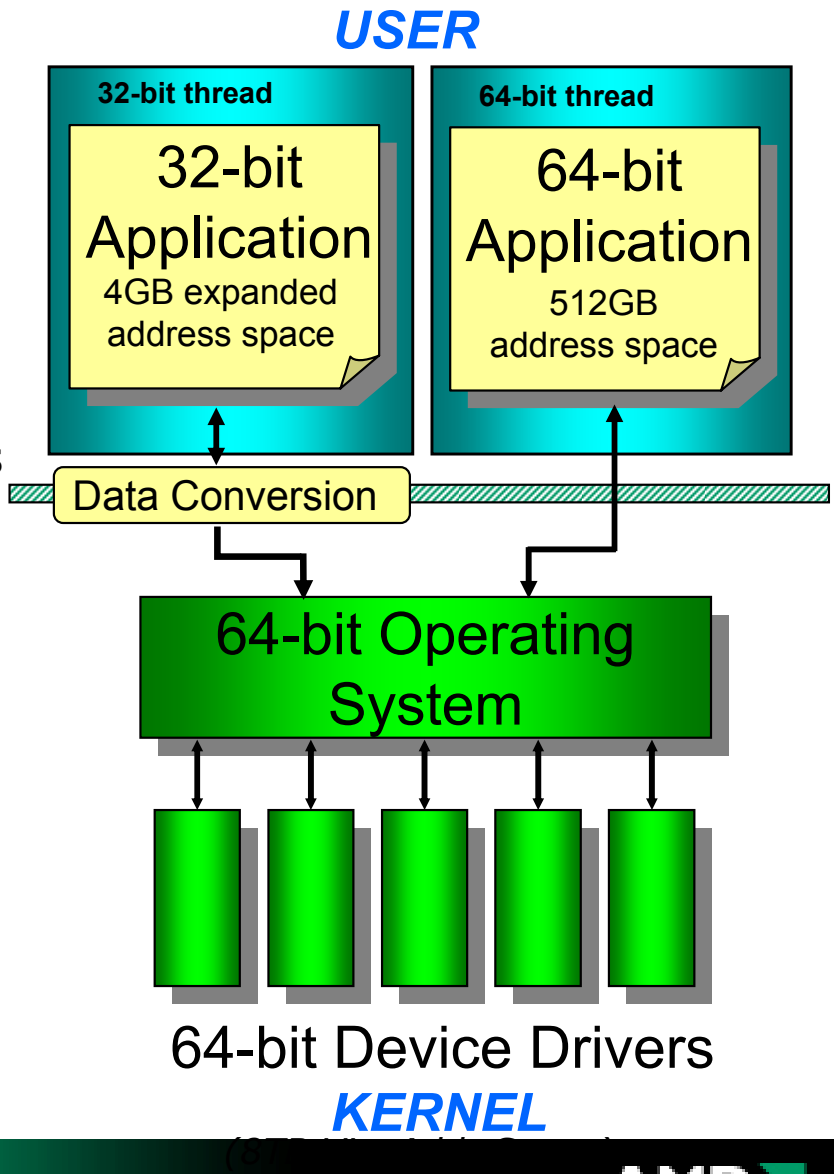
64-bit OS & Application Interaction

32-bit Compatibility Mode

- 64-bit OS runs existing 32-bit APPs with leading edge performance
- No recompile required, 32-bit code directly executed by CPU
- 64-bit OS provides 32-bit libraries and "thunking" conversion layer for 32-bit system calls.
- Thunking layer implements all 32-bit kernel calls
  - Converts parameters as necessary
  - Calls 64-bit kernel
  - Converts results as necessary

64-bit Mode

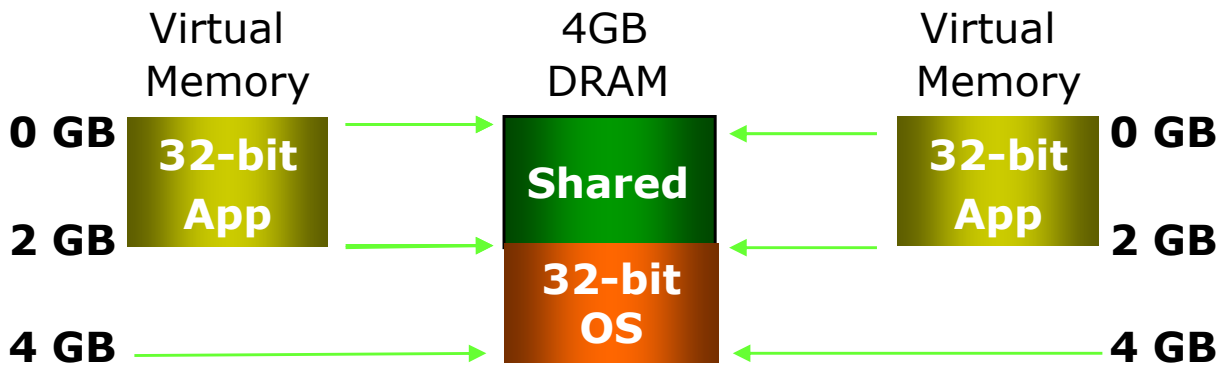
- Migrate only where warranted, and at the user's pace to fully exploit AMD64
- 64-bit OS requires all kernel-level programs & drivers to be ported.
- Any program that is linked or plugged in to a 64-bit program (ABI-level) must be ported to 64-bits.



Increased Memory for 32-bit Applications

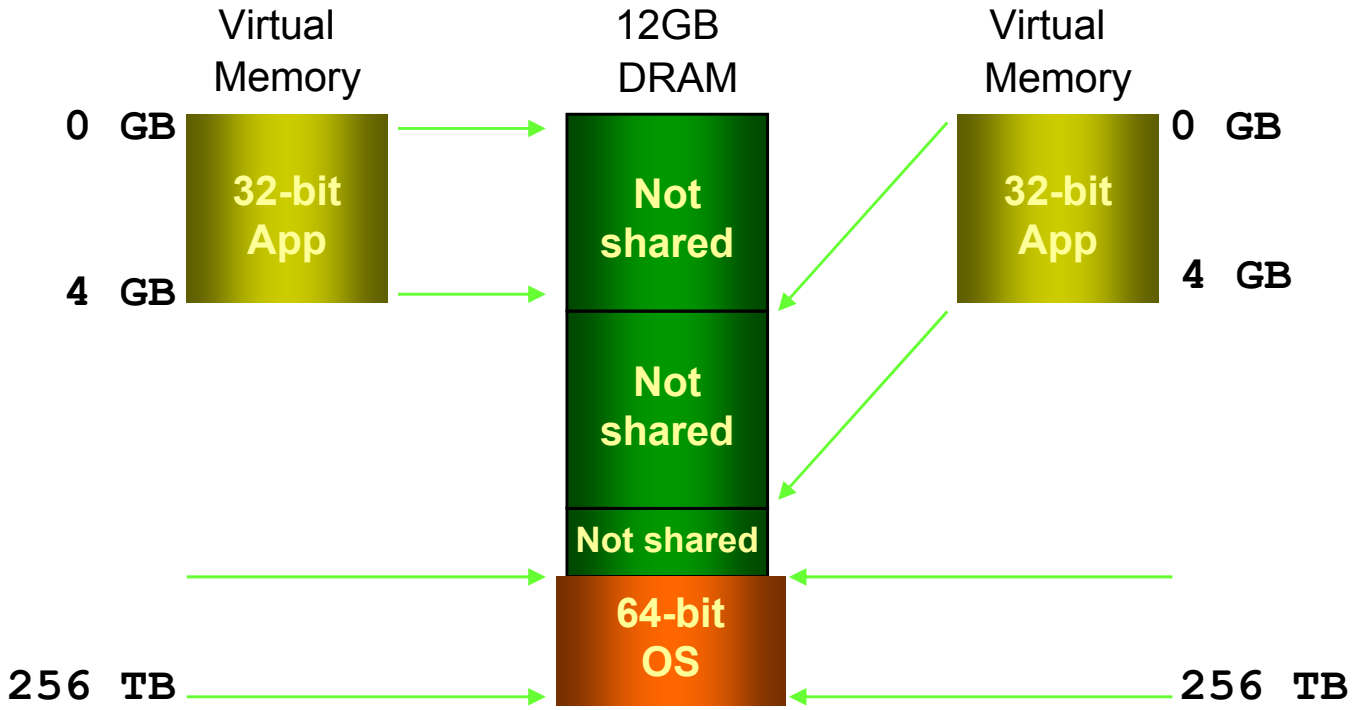
32-bit server, 4 GB DRAM

- OS & App share small 32-bit VM space
- 32-bit OS & applications all share 4GB DRAM
- Leads to small dataset sizes & lots of paging



64-bit server, 12 GB DRAM

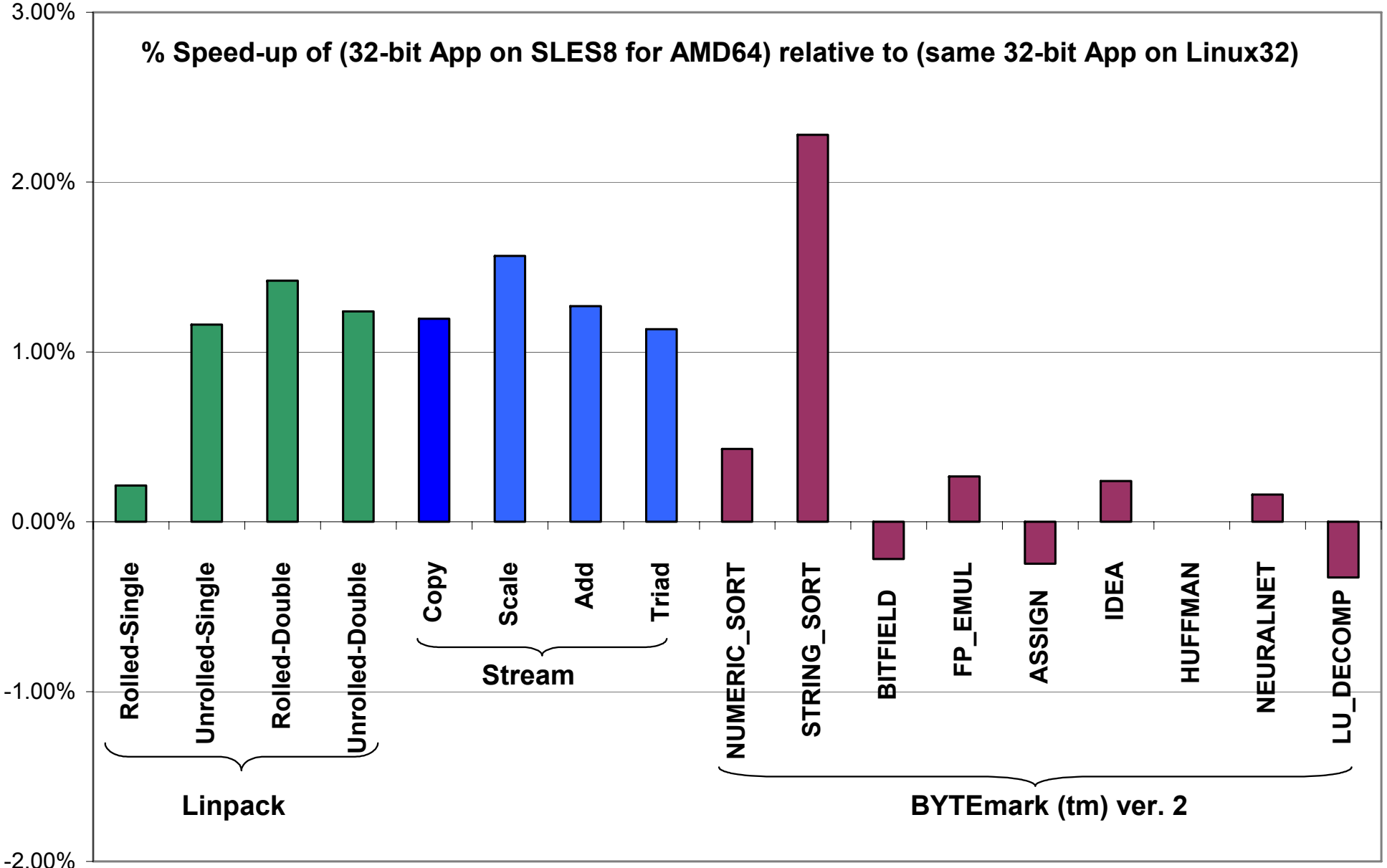
- App has exclusive use of 32-bit VM space
- 64-bit OS can allocate each application large dedicated portions of 12GB DRAM
- OS uses VM space way above 32-bits
- Leads to larger dataset sizes & reduced paging



x86 32-bit Binaries on AMD64 Linux

- Work just fine and at full speed (no simulation, no emulation).
- System Call Thunking provides the necessary infrastructure.
  - Using 64-bit libs requires inter-process communication and is not preferred route.
- 32-bit libraries remain in */lib.
- Same x86 32-bit SYSV ABI. Same 32-bit development tools.
  - Floating-point model is x87.
  - SSE, SSE2, MMX, and 3Dnow are supported
  - Inline asm still works, etc …
- 64-bit OS is more efficient & faster than 32-bit OS on same HW, thus total solution of 64-bit OS & 32-bit app will run faster.

32-bit App Performance on 64-bit Linux



Measured on AMD Opteron Model 144 (1.8GHz, 1MB L2, 128-bit Memory Controller, DDR-333, CL 2.5, 512MB)

Trademark Attribution

The Opteron/Athlon64 Optimization Guide

- Opteron/Athlon64 Optimization Guide is available at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF
- Audience is both code generator designers/assembly level programmers and C/C+ level programmers
  - emphasis on code generation/assembly level.
- Most optimizations apply to both 64-bit and 32-bit code.
- Note: There is also an overview of the Micro Architecture in Appendix A

The Opteron/Athlon64 Optimization Guide

- Optimizations organized in Micro-Architecture Sections
  - Instruction Decoding Optimizations
  - Cache and Memory Optimizations
  - Branch Optimizations
  - Scheduling Optimizations
  - Integer Optimizations
  - SIMD Optimizations
  - x87 Optimizations
- A great feature is “Key Optimizations” which lists 14 most important optimizations, cutting across all of the above

Some Key Optimizations

- Avoid Store to Load Forwarding Stalls
  - For example, storing 64 bits, loading back the high 32 bits
  - Processor can usually handle load forwarding from store buffers, but not when low address bits differ.
  - In those cases, load must wait for store to retire and be written to the data cache

    ie, every preceding instruction must also retire

- Data Alignment
  - For most instructions, the processor handles unaligned data without traps, but there are performance penalties

The Opteron/Athlon64 Optimization Guide

- Watch structure attribute ordering and packing.
- Most efficient structure packing achieved by placing largest members first, followed by next largest, etc…
- struct s1 {
  char c;
  int i;
  char *p;
  long n;
  }

  struct s2 {
  char *p;
  long n;
  int i;
  char c;
  }
- struct s1 takes 24 bytes with 4 bytes padding after ‘i’, s2 takes 21 bytes with no padding.

Some Key Optimizations (continued)

- Branch Density
  - The processor's branch prediction logic is limited to 3 branches per 16-byte window.
  - Avoid branches that cross 16-byte boundaries when possible
- RET instruction
  - A two-byte near-return RET instruction avoids some branch prediction restrictions which could affect a single-byte RET
- Align Branch Targets in Program Hot Spots to 16-byte boundaries
  - maximizes decoding bandwidth
  - preserves I-cache space

Some Key Optimizations (continued)

- Prefetch Instructions
  - There is a hardware prefetcher which handles many simple prefetching scenarios
  - But Software Prefetch can still be useful
  - Prefetch Instruction will prefetch into L1, Hardware Prefetch into L2
  - Hardware Prefetch limited to detecting consecutive cache block accesses.
  - Software Prefetch allows finer control

    PREFETCH for blocks that will only be read or that you are not sure will be written

    PREFETCHW for block that you know will be written

    PREFETCHNTA for "non-temporal", use for data that will not be needed again

Some Key Optimizations (continued)

- Use DirectPath Instructions
  - Covers most frequently used instructions
  - VectorPath = microcoded, only 1 can be decoded per cycle
  - Optimization Guide Appendix C lists all instructions, their result latencies and whether they are DirectPath or VectorPath
- Use Load-Execute Instructions
  - In general, they are DirectPath
  - Using separate Load and Execute instructions reduces decoding bandwidth and can increase register pressure

Some Key Optimizations (continued)

- Avoid Data-Dependent Branches where possible
  - branch predictor will be wrong 50% of the time
  - CMOV (Conditional Move) instruction can help in some cases

- Avoid Placing Code and Data in the same 64-byte Cache Line
  - can cause thrashing to maintain coherency in the separate instruction and data caches
  - for example, JMP table data should be separated from the code containing the JMP instruction

Some Key Optimizations (continued)

- Avoid SSE/SSE2 Data Type Mismatch
  - SSE/SSE2 instructions expect and produce either
    Single Precision Floating Point
    Double Precision Floating Point
    Integer
  - Performance can decrease if input register to an instruction is not the expected data type
  - Good News, Logical Instructions and Load/Store are universal

Some 64-Bit Specific Issues

- 64-bit Register Addressing
  - only 64-bit GPRs can be used in addressing modes
  - signed 32-bit offsets need to be sign-extended to 64 bits
  - Look for opportunities to hoist such sign extension out of loops
- Use 64-bit registers for 64-bit integer arithmetic including 64-bit multiplication

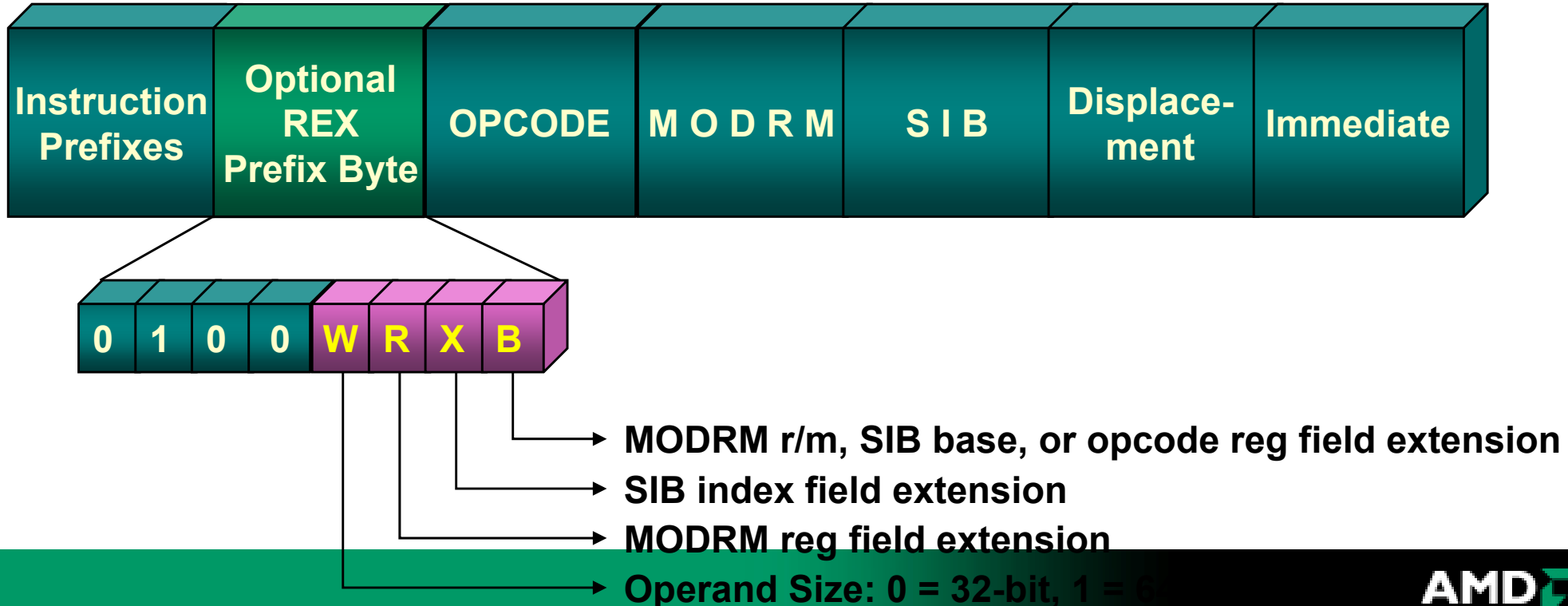
64-Bit Mode Operation

- Default data size is 32 bits
  - Override to 64 bits using new REX prefix
  - Override to 16 bits using legacy operation size prefix (66h)
- Default address size is 64 bits
  - Pointers are 64 bits
- 2 New instructions added, Some redundant encodings reclaimed
  - MOVSXD: Move sign extended double to quad
  - SWAPGS: Allows quick swap of GS in ISRs
- New override (REX) allows naming 16 GP and 16 SSE registers
  - Only 1 override byte per-instruction is needed for extended registers; regardless of how many are used by the instruction

Prefix Type	Default	REX	66h
Operand Size	32	64	16

64-bit Mode Opteration: REX Prefix Byte

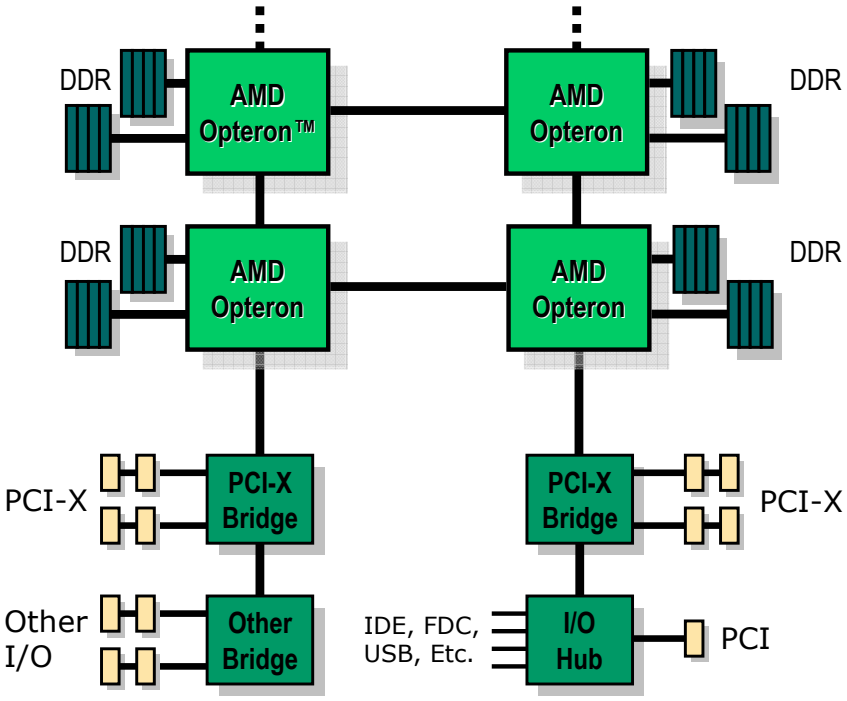
- Optional REX prefix specifies 64-bit operation size override and 3 additional register encoding bits
  - Extra registers encoded without altering existing instruction format
  - REX is actually a family of 16 prefixes (40-4F)
  - 64-bit mode Average instruction length increased by 0.4 bytes



Advanced AMD Opteron™ Processor

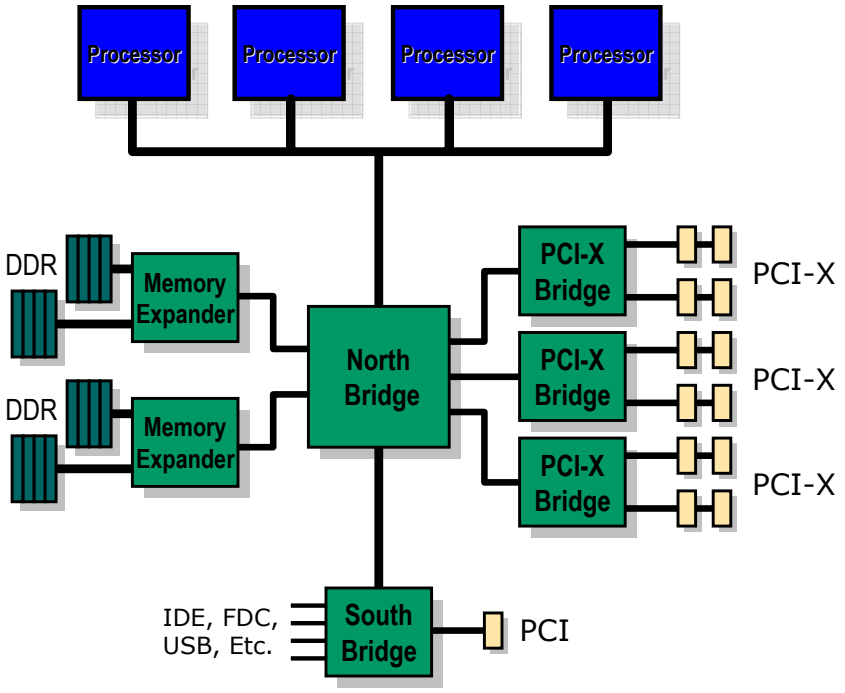
Multiprocessor system architecture

AMD Opteron™ Processor-based System



- Up to 8 processors without glue logic
- Each processor adds memory
- Each processor adds additional HyperTransport™ buses for PCI-X and other I/O bridges
- Fewer chips required

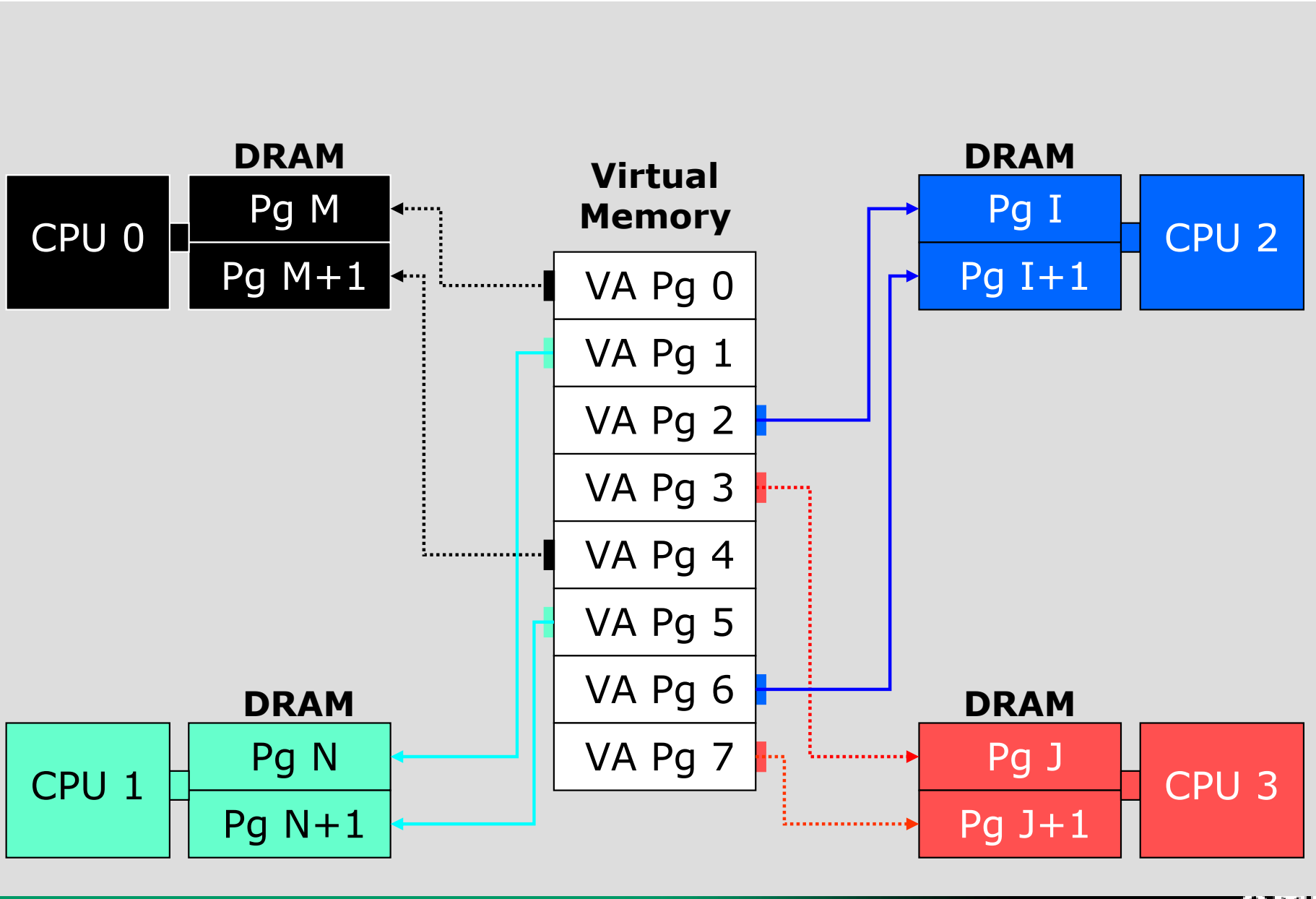
Typical MP System



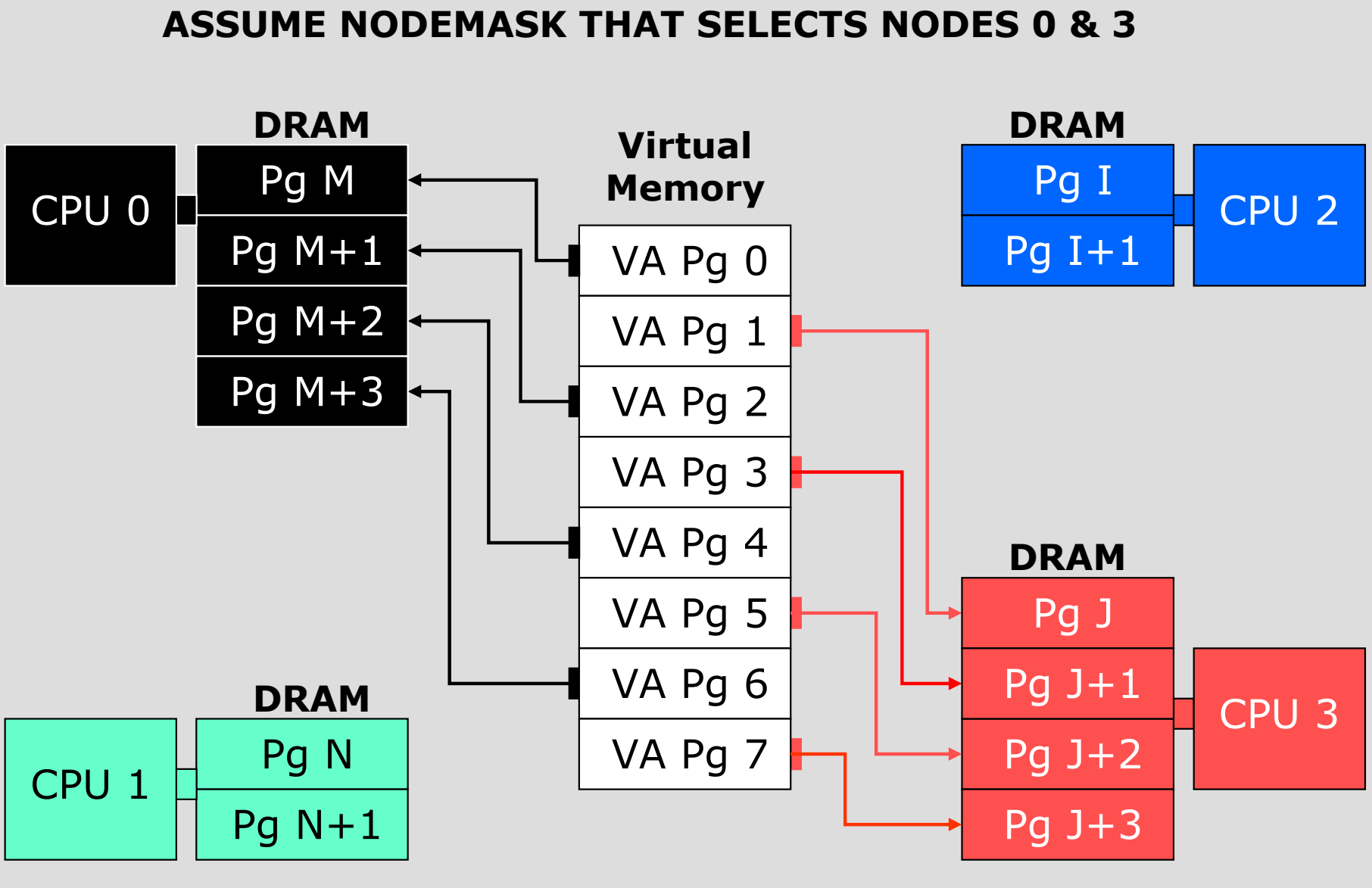
- Processors compete for FSB bandwidth
- Memory size and bandwidth are limited
- Maximum of 3 PCI-X bridges
- More chips required

NUMA Support in the Linux Kernel for AMD64

Interleaved Memory Alloc



Interleaved Memory Alloc with Mask



Example: Interleaved Numa Memcpy (1)

```
#include <pthread.h>
#include <numa.h>
< . . . >
void *copythread(void *argp) {
     struct copyargs *args = argp;
     numa_run_on_node(args->node);
     atomic_dec(&notrunning);
     /* spin on lock until we can start */
     while (start_lock == 0);
     do_copy(args);
     atomic_dec(&notfinished);
     return NULL;
}
< . . . >
```

Example: Interleaved Numa Memcpy (2)

```
int main(void){
   void *mem;
   < . . . >
   if (numa_available() < 0) { exit(1);}
   max_node = numa_max_node();
   stride_init = numa_get_interleave_node();
   mem = numa_alloc_interleaved(size);
   if (!mem) exit(1);
 numa_run_on_node(stride_init);
   < . . . >
   pthread_t thr[max_node];
   struct copyargs args[max_node];
   notrunning = max_node - 1;
   notfinished = max_node - 1;
   stride = stride_init;
   < . . . >
```

Example: Interleaved Numa Memcpy (3)

```
for (i = 0; i < max_node; i++) {
    args[i].node = stride;
    stride = (1+stride)% max_node;
    <Setup Src, Dst, Size for interleaved Copy>
    if (i != stride_init)
       pthread_create(&thr[i], NULL, copythread, &args);
}
/* Wait for the threads to startup */
    while (notrunning > 0) ;
/* Start the copy -- let others start */
    start_lock = 0;
    do_copy(&args[0]);
/* wait for others */
    while (notfinished > 0) ;
    <Join Threads . . . >
numa_free(mem, size);
   <return>
```

BACKUP: MEMCPY Program using libnuma (1)

```
#include <pthread.h>
#include <numa.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MB (1024*1024)
int max_node;
long size = 300 * MB;

struct copyargs {
     char *src;
     char *dst;
     int size;
     int node;
     int blocksize;
     int step;
};

volatile unsigned start_lock = 1;
volatile unsigned notrunning;
volatile unsigned notfinished;
```

```
static inline unsigned long rdtsc(void)
{
     unsigned low, high;
     asm volatile("rdtsc" : "=a" (low), "=d"
     (high));
     return ((unsigned long)high << 32) | low;
}

static inline void atomic_dec(volatile unsigned
     *val)
{
     asm("lock ; decl %0" : "+m" (*val));
}

static inline void do_copy(struct copyargs *args)
{
     int n = 0;
     int blocksize = args->blocksize;
     int step = args->step;
     char *src = args->src;
     char *dst = args->dst;
     while (n < args->size) {
             memcpy(dst, src, blocksize);
               /* inline would be better */
             src += step;
             dst += step;
             n += blocksize;
     }
}
```

BACKUP: MEMCPY Program using libnuma (2)

```
void *copythread(void *argp)
{
     struct copyargs *args = argp;
     numa_run_on_node(args->node);
     atomic_dec(&notrunning);
     /* spin on lock until we can start */
     while (start_lock == 0);
     do_copy(args);
     atomic_dec(&notfinished);
     return NULL;
}

int main(void)
{
     void *mem;
     int i, stride, stride_init;
     unsigned long start,end;
     int page_size = getpagesize();

     if (numa_available() < 0) {
            fprintf(stderr, "Your kernel doesn't
                        support NUMA policy.\n");
            exit(1);
     }
     max_node = numa_max_node();
     stride_init = numa_get_interleave_node();
     mem = numa_alloc_interleaved(size);
     if (!mem) exit(1);
```

```
numa_run_on_node(stride_init);
memset(mem, 0xff, size/2);
memcpy(mem+size/2, mem, size/2);
/* prime the dynamic linker on memcpy */

pthread_t thr[max_node];
struct copyargs args[max_node];
notrunning = max_node - 1;
notfinished = max_node - 1;
stride = stride_init;
int offset = 0;
for (i = 0; i < max_node; i++) {
        args[i].node = stride;
        stride = (1+stride)%max_node;
        args[i].step = page_size * max_node;
        args[i].blocksize = page_size;
        args[i].src = mem + offset;
        args[i].dst = mem + size/2 + offset;
        args[i].size = (size/2)/max_node;
        offset += page_size;
        if (i != stride_init)
                    pthread_create(&thr[i],
                    NULL, copythread, &args);

}
/* Wait for the threads to startup */
while (notrunning > 0) ;
```

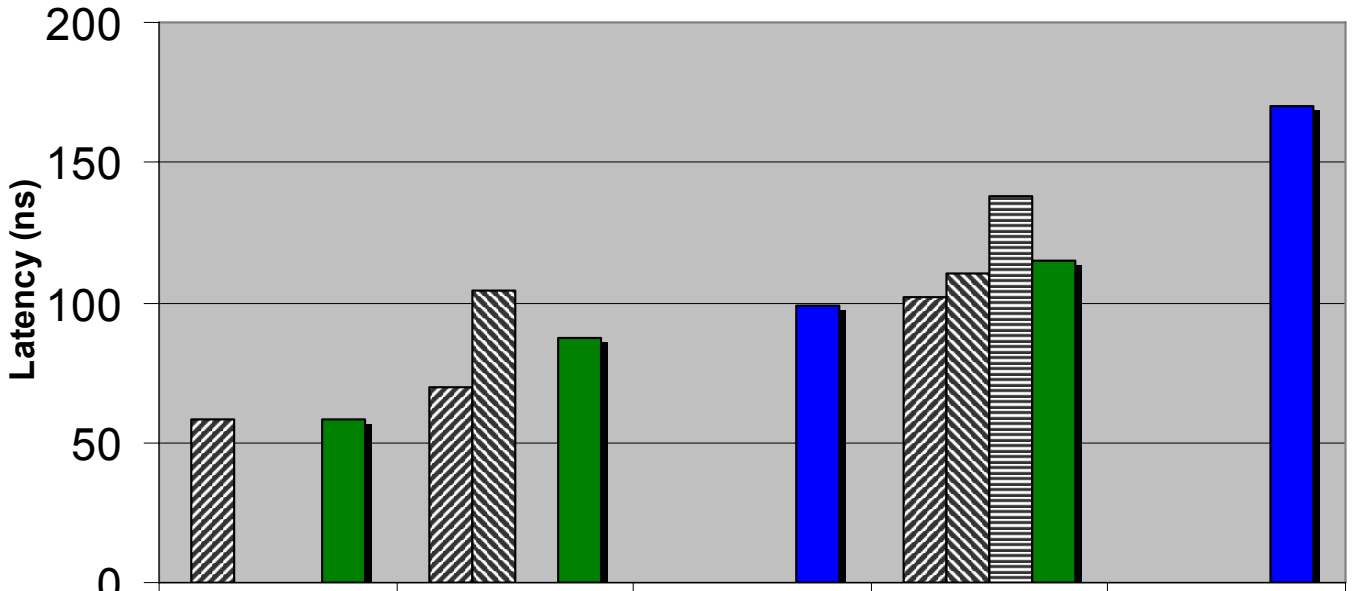
BACKUP: MEMCPY Program using libnuma (3)

```
/* Start the copy */
start = rdtsc();
start_lock = 0;     /* let others start */
do_copy(&args[0]);
while (notfinished > 0) /* wait for others */
       ;
end = rdtsc();
for (i = 0; i < max_node; i++)
       pthread_join(thr[i], NULL);

printf("interleaved %ld cycles (%.3f/MB)\n",
       end-start, (float)(end-start)/MB);

args[0].src = mem;
args[0].dst = mem+size/2;
args[0].step = page_size;
args[0].size = size/2;
args[0].blocksize = page_size;
start = rdtsc();
do_copy(&args[0]);
end = rdtsc();
printf("single threaded %ld cycles
(%.3f/MB)\n",
       end-start,(float)(end-start)/MB);
numa_free(mem, size);
```

```
mem = numa_alloc_local(size);
args[0].src = mem;
args[0].dst = mem+size/2;
args[0].step = page_size;
args[0].size = size/2;
args[0].blocksize = page_size;
start = rdtsc();
do_copy(&args[0]);
end = rdtsc();
printf("local %ld cycles (%.3f/MB)\n",
       end-start, (float)(end-start)/MB);
numa_free(mem, size);

return 0;
```

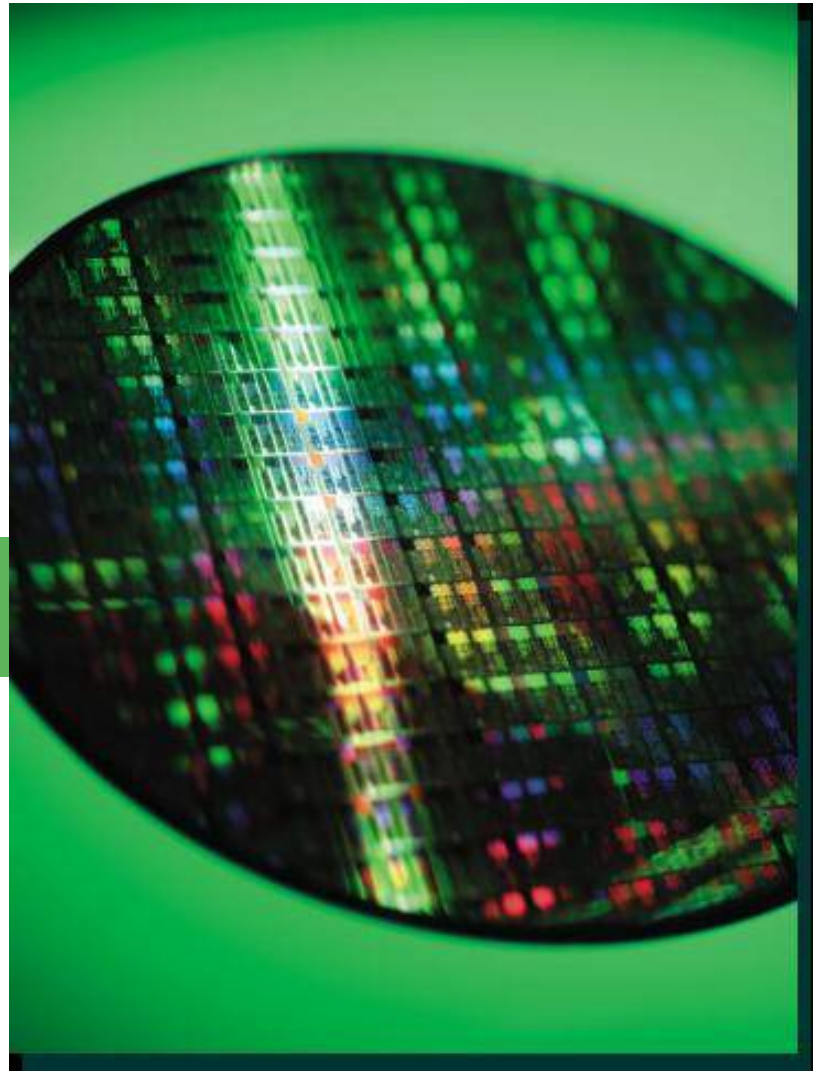
Memory Latency

ScienceMark 2.0 Beta,
512-Byte Stride Latency (ns)



	1 AMD Opteron™ Model 146 2xPC2700 CL2.5	2 AMD Opteron Model 246 4xPC2700 CL2.5	1 P4 3.0GHz, 800 FSB, 875, 4xPC3200 CL2	4 AMD Opteron Model 846 8xPC2700 CL2.5	Dell PE 6600 4 XeonMP 2GHz 16xPC1600 CL2
0-hop (512 byte stride, ns)	58	70		102	
1-hop (512 byte stride, ns)		104		110	
2-hop (512 byte stride, ns)				138	
Weighted Average	58	87	99	115	

Linux AMD64 Tricks



AMD64 Programmer's View

How to tell a 64-bit App from a 32-bit App

- rpm -qia | grep -i i386
  - Not many 32-bit apps in this distribution …
- Cat /proc/<pid>/maps
  - If the map references "lib64", then it is a 64-bit app
  - If the map doesn't reference "lib64", then it is a 32-bit app.
- uname -m
  - Returns "x86_64" normally.
  - With linux32, can return "i686"

32-bit Thread

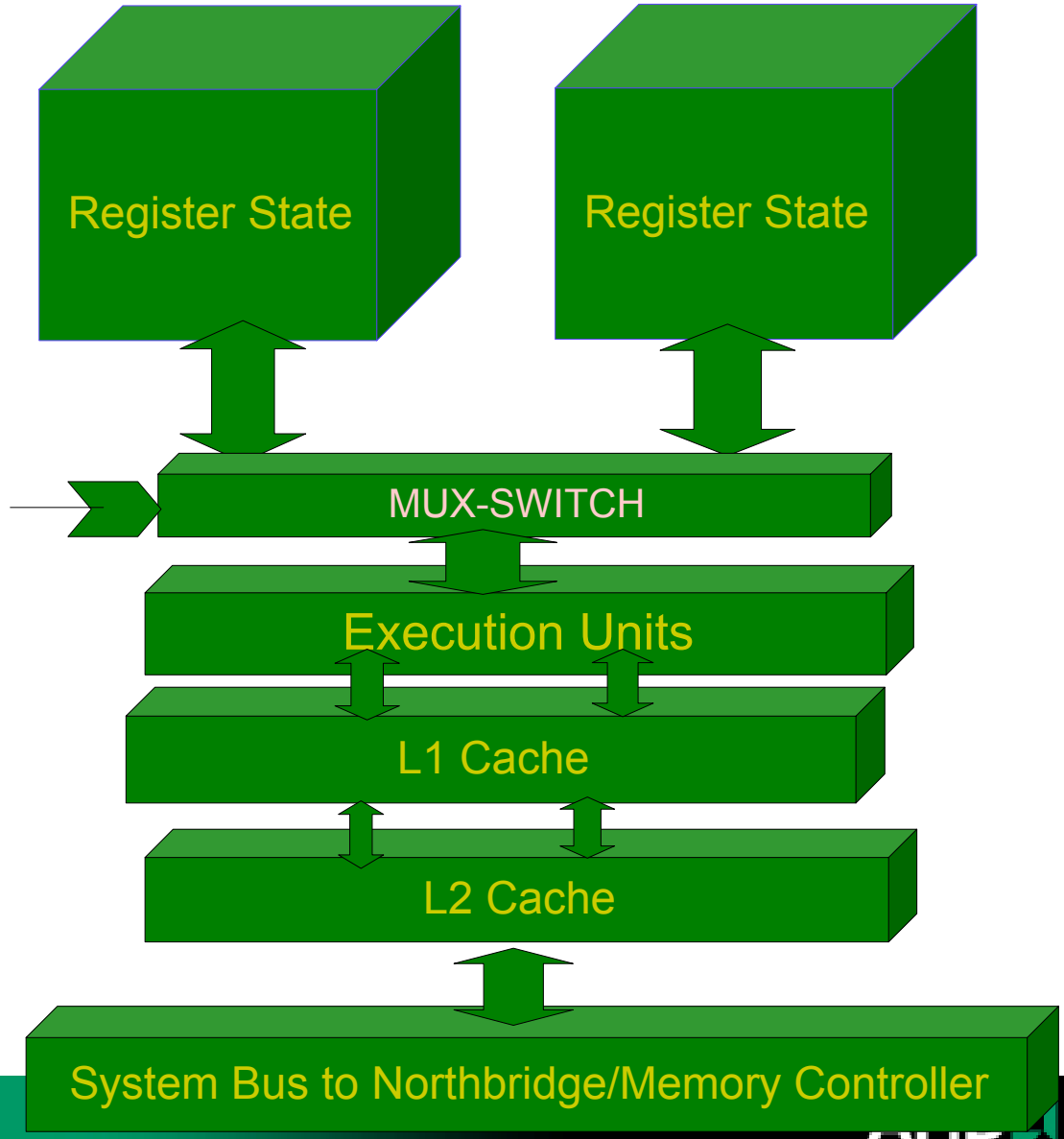
```
% cat /proc/`ps --no-headers -o pid -C
   duh32`/maps

00000000008048000-00000000008049000 r-xp 0000000000000000 03:03
   136399     /home/obrien/proj/porting/duh32
00000000008049000-0000000000804a000 rwxp 0000000000000000 03:03
   136399     /home/obrien/proj/porting/duh32
00000000040000000-00000000040013000 r-xp 0000000000000000 03:03
   28949      /lib/ld-2.2.5.so
00000000040013000-00000000040014000 rwxp 0000000000012000 03:03
   28949      /lib/ld-2.2.5.so
00000000040014000-00000000040015000 rwxp 0000000000000000 00:00 0
0000000004002b000-00000000040141000 r-xp 0000000000000000 03:03
   28955      /lib/libc.so.6
00000000040141000-00000000040147000 rwxp 0000000000115000 03:03
   28955      /lib/libc.so.6
00000000040147000-0000000004014b000 rwxp 0000000000000000 00:00 0
00000000ffffd000-00000000fffff000 rwxp ffffffffffffe000 00:00 0
```

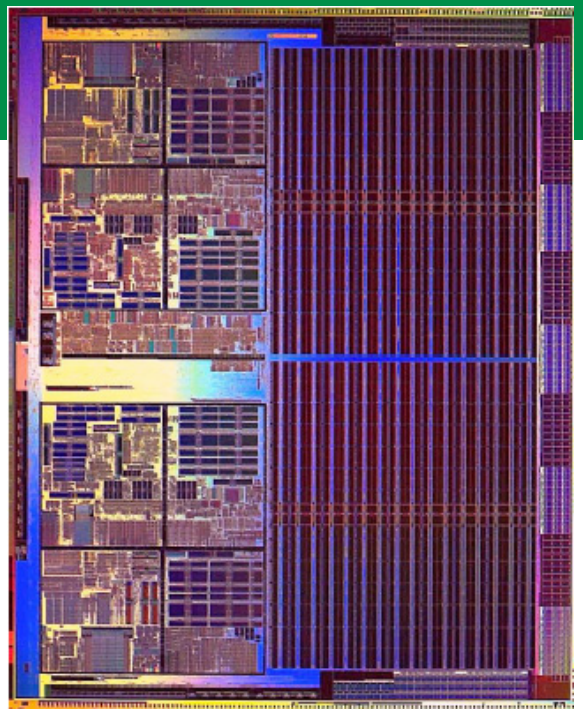
64-bit Thread ...

```
% cat /proc/`ps --no-headers -o pid -C duh64`/maps
0000000000400000-0000000000401000 r-xp 0000000000000000 03:03 136400
   /home/obrien/proj/porting/duh64
0000000000500000-0000000000501000 rw-p 0000000000000000 03:03 136400
   /home/obrien/proj/porting/duh64
0000002a95556000-0000002a95568000 r-xp 0000000000000000 03:03 20316
   /lib64/ld-2.2.5.so
0000002a95568000-0000002a9556a000 rw-p 0000000000000000 00:00 0
0000002a95667000-0000002a9566a000 rw-p 0000000000011000 03:03 20316
   /lib64/ld-2.2.5.so
0000002a9566a000-0000002a95773000 r-xp 0000000000000000 03:03 20321
   /lib64/libc.so.6
0000002a95773000-0000002a9586a000 ---p 0000000000109000 03:03 20321
   /lib64/libc.so.6
0000002a9586a000-0000002a95891000 rw-p 0000000000100000 03:03 20321
   /lib64/libc.so.6
0000002a95891000-0000002a95897000 rw-p 0000000000000000 00:00 0
0000007fbfffe000-0000007fc0000000 rwxp fffffffffffff000 00:00 0
```

How Does AMD Dual Core CPU's Differ From Hyper Threaded (HTT) CPU's?

- Hyper Threaded CPU's maintain 2 register contexts-(threads)
- But each context will share same execution resources (integer, FPU, Address gen. units) and also L1, and L2 caches
- In reality, **only** 1 register state can execute at any one time. I.E. there is NO true multiprocessing occurring





Developing for the AMD™64 Processor & Platform Overview

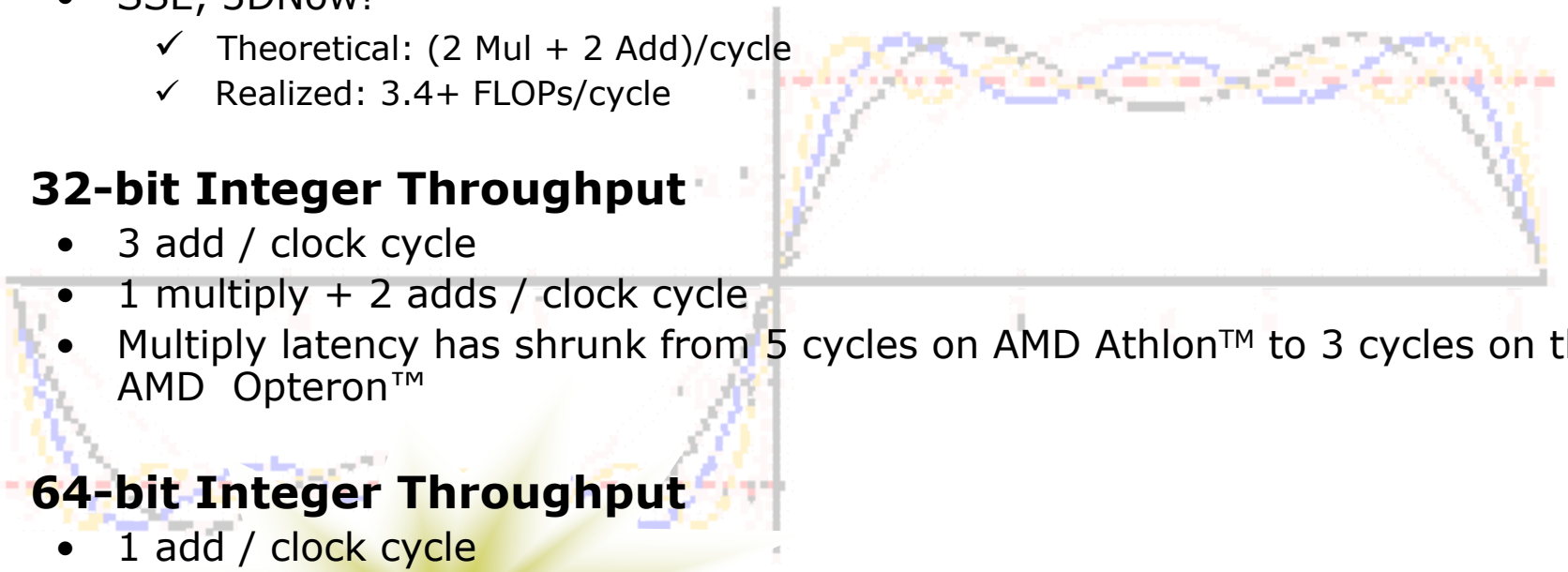
David O'Brien
Sr. Systems Software Engineer
Advanced Micro Devices

Cost of cache misses

- L1 Cache Hit: 3 cycles
- L1 Miss/L2 Hit: 20 cycles
  - 3 cycle L1 cache latency, miss detection
  - 4 cycles to write out a modified cache line (16 bytes/cycle)
  - L2 cache itself has 9 cycle latency
  - 4 cycles to transfer new cache line to L1 (16 bytes/cycle)
- L1 Miss/L2 Miss/DRAM: 70-90 cycles

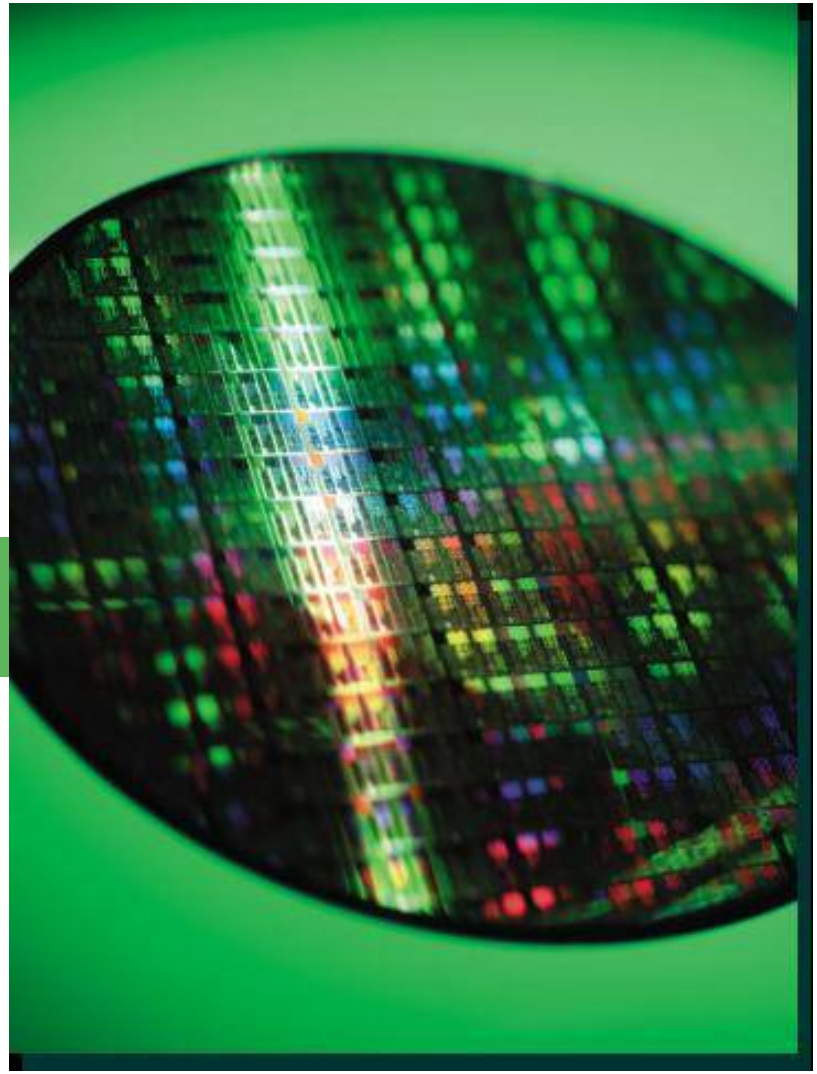
Floating Point & Integer Performance

- **FPU Throughput**
  - SSE2, x87
    - ✓ Theoretical: (1 Mul + 1 Add)/cycle
    - ✓ Realized: 1.9 FLOPs/cycle
  - SSE, 3DNow!
    - ✓ Theoretical: (2 Mul + 2 Add)/cycle
    - ✓ Realized: 3.4+ FLOPs/cycle



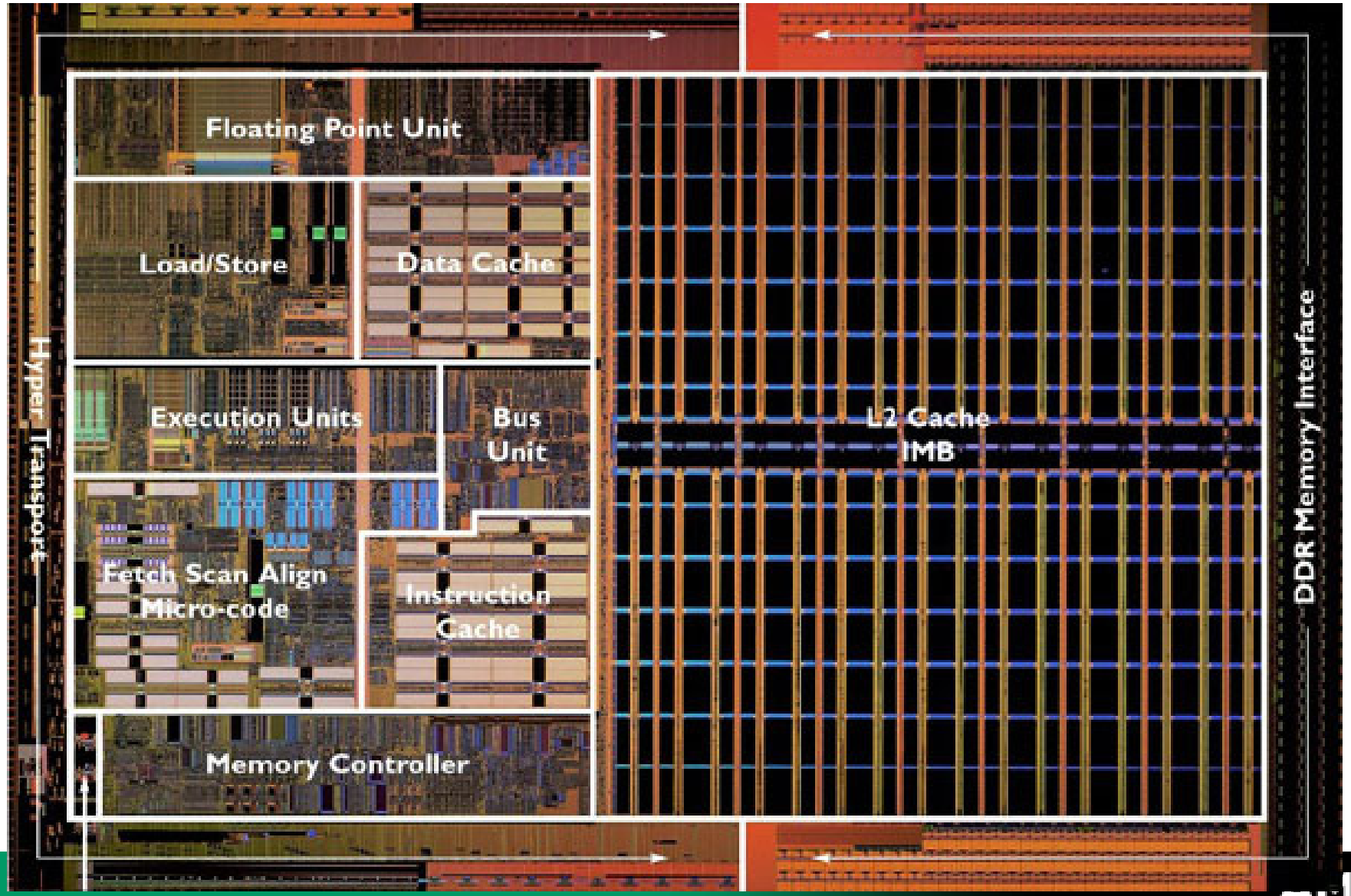
- **32-bit Integer Throughput**
  - 3 add / clock cycle
  - 1 multiply + 2 adds / clock cycle
  - Multiply latency has shrunk from 5 cycles on AMD Athlon™ to 3 cycles on the AMD Opteron™
- **64-bit Integer Throughput**
  - 1 add / clock cycle
  - 1 multiply every other clock cycle
  - Multiply latency is 4 cycles
  - Integer Instruction Scheduler
    - ✓ Out Of Order (OOO) from a queue of 24* Integer Macro-Ops

**Athlon™ Instruction Scheduler is 18 Macro-Ops deep*

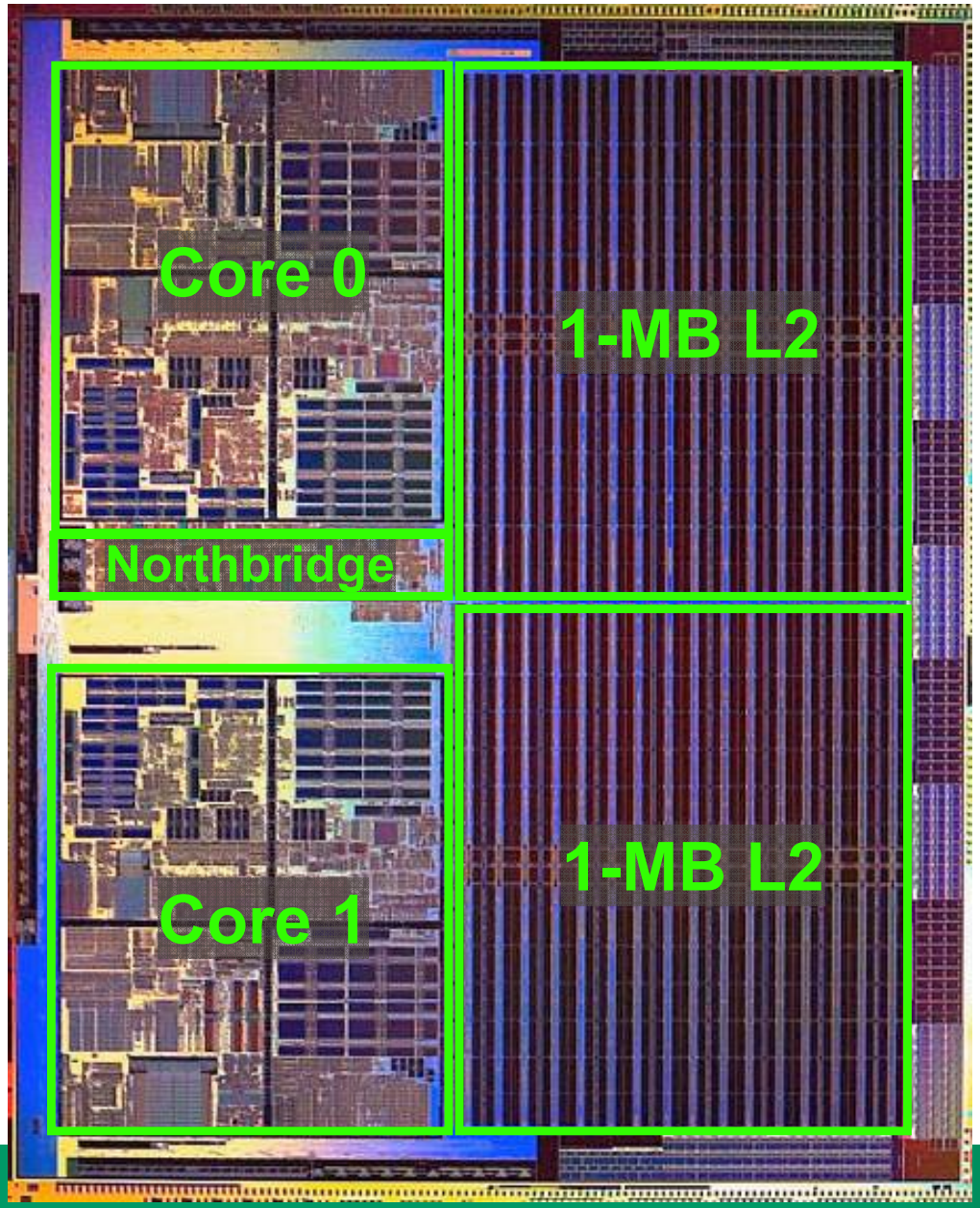


AMD64 Building Blocks

The AMD Opteron™ Processor Layout



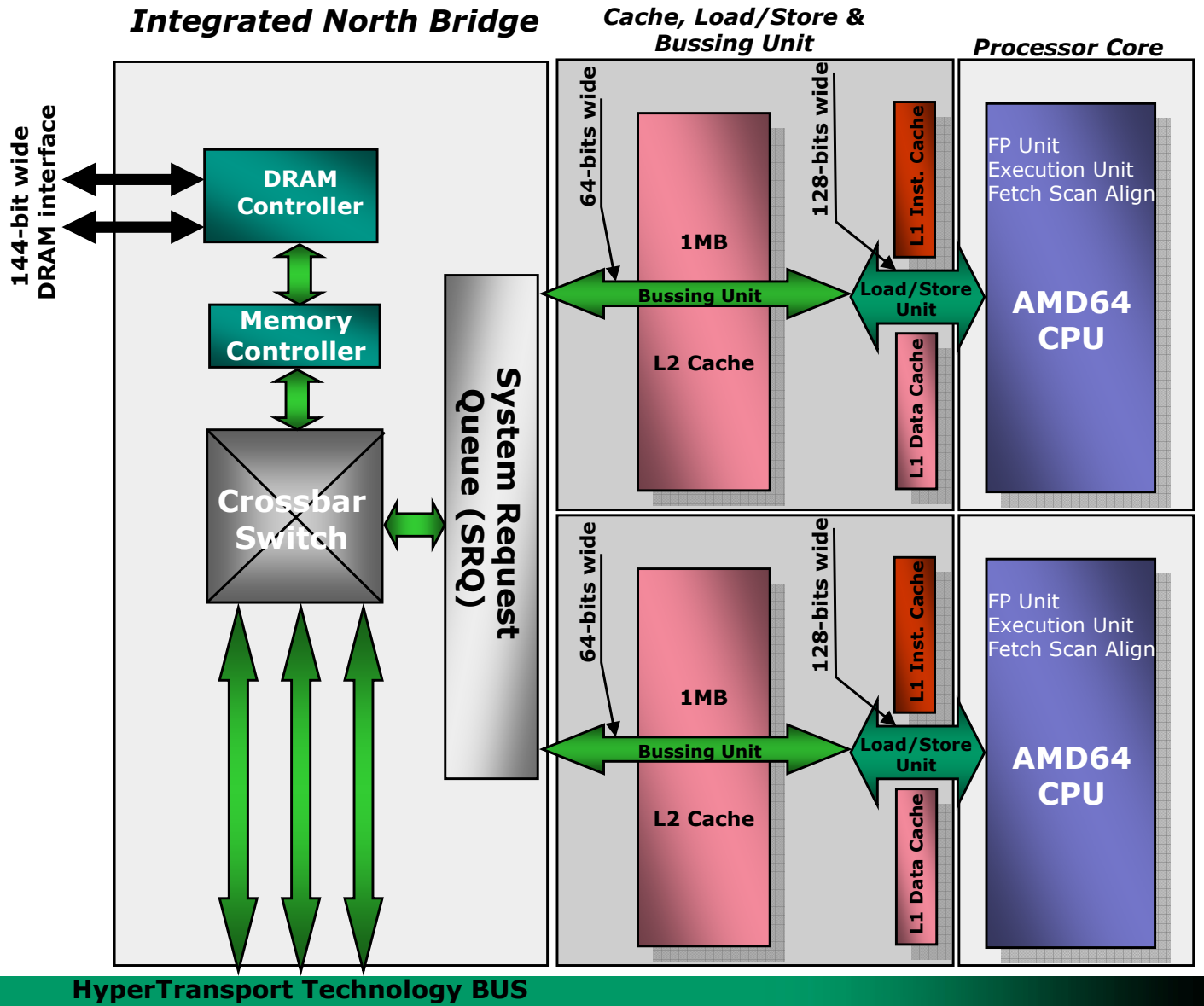
The AMD64 Dual-Core Processor



- Two AMD Opteron™ CPU cores on a single die, each with 1MB L2 cache
- 90nm, ~205 million transistors*
  - Approximately same die size as 130nm single-core AMD Opteron processor*
- Near MP scaling
- Dual-Core is not Hyper-Threading
  - Dual-Core has no conflicts for cache space
  - No collisions in FPU or ALU pipelines
  - No conflict for I-cache or D-cache access ports
  - No sharing of branch prediction units
  - Compatibility: Dual-Core runs legacy Hyper-Threaded code
    - *CPU ID feature detection bits indicate "HTT capability"*
    - *But Hyper-Threaded code might not fully utilize both cores*
- Introduced with "K8" Revision E core in April 2005

*Based on current revisions of the design

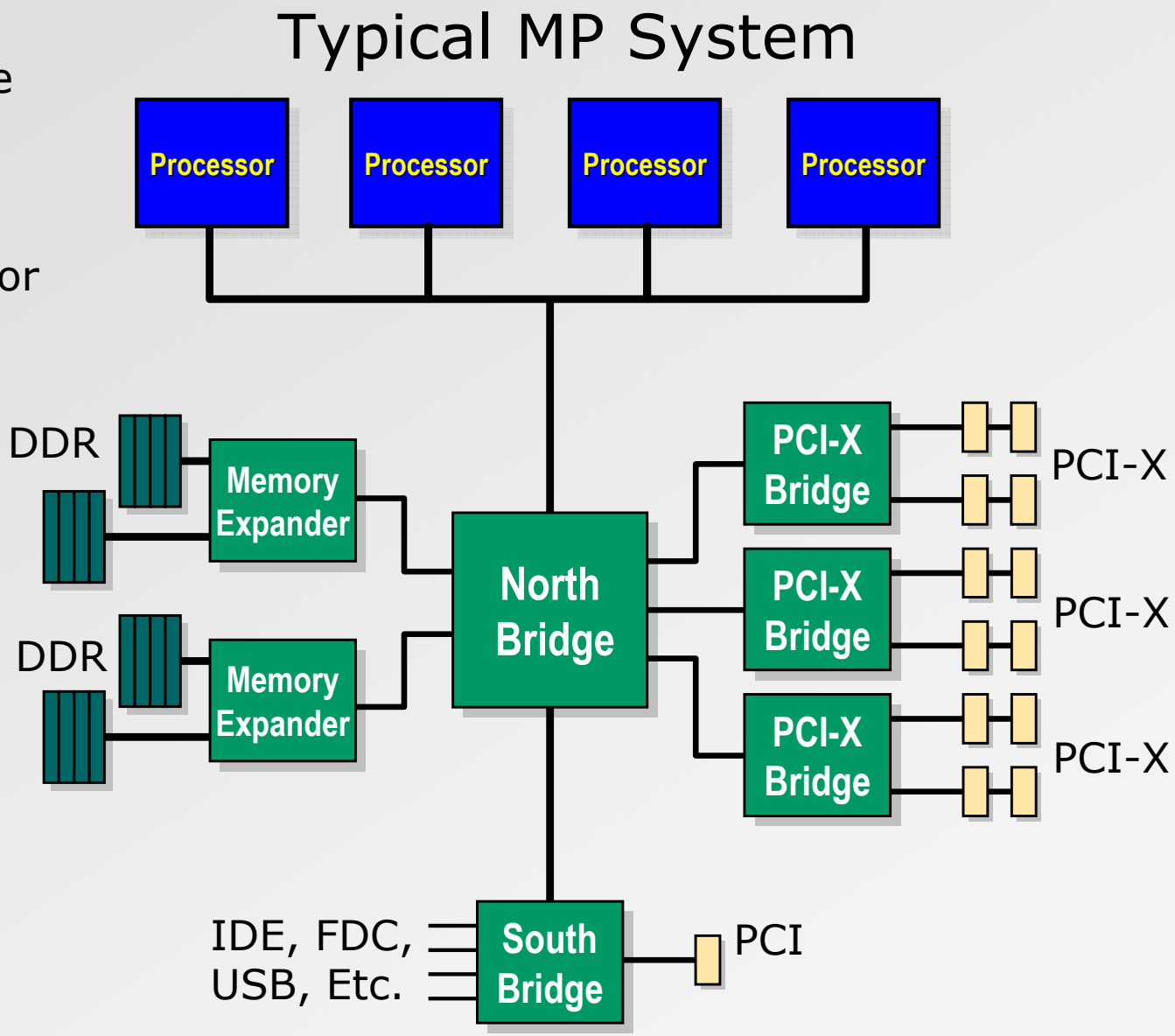
Dual Core-Data Flow



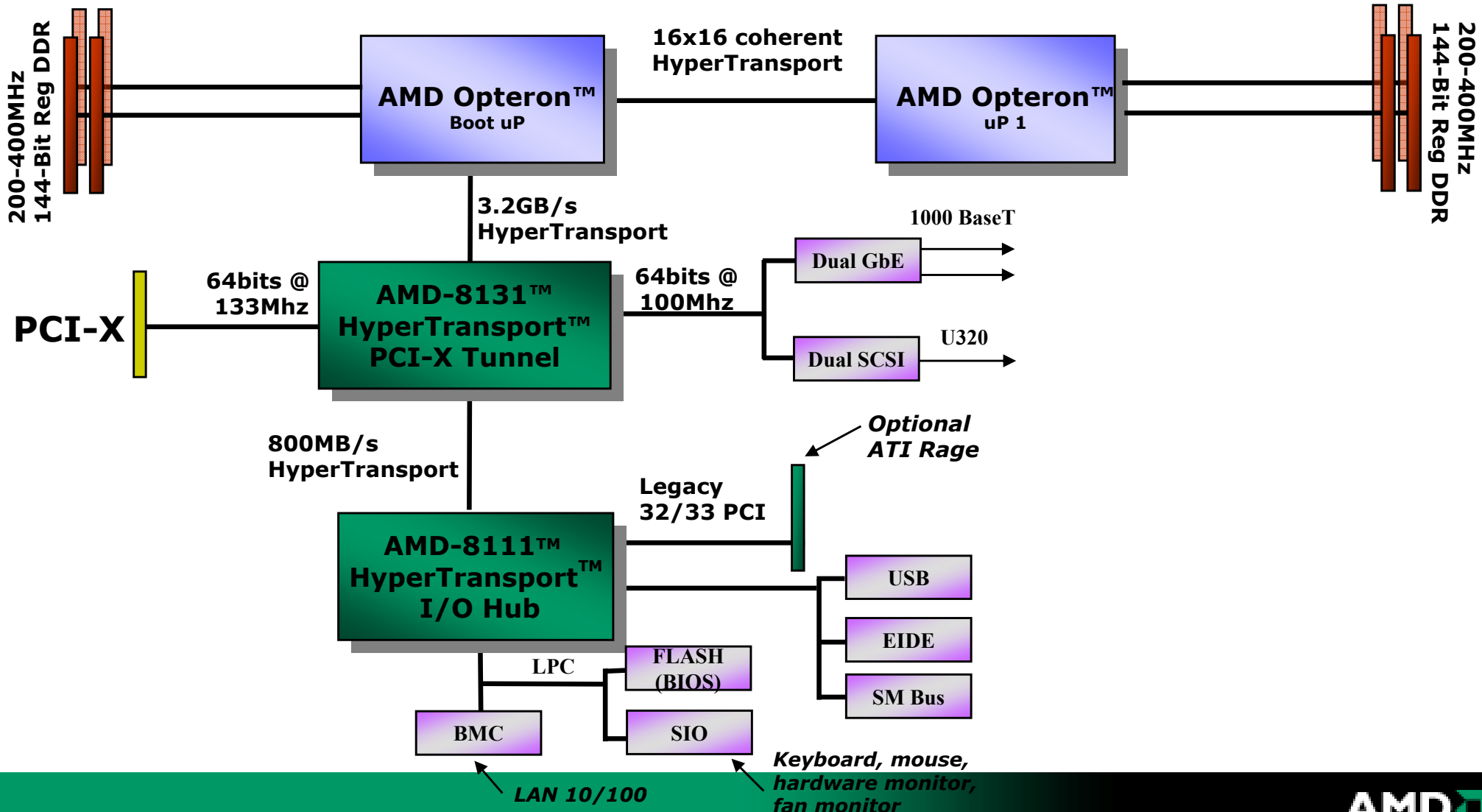
The K8 dual port interface (SRQ)

Typical Multiprocessing System

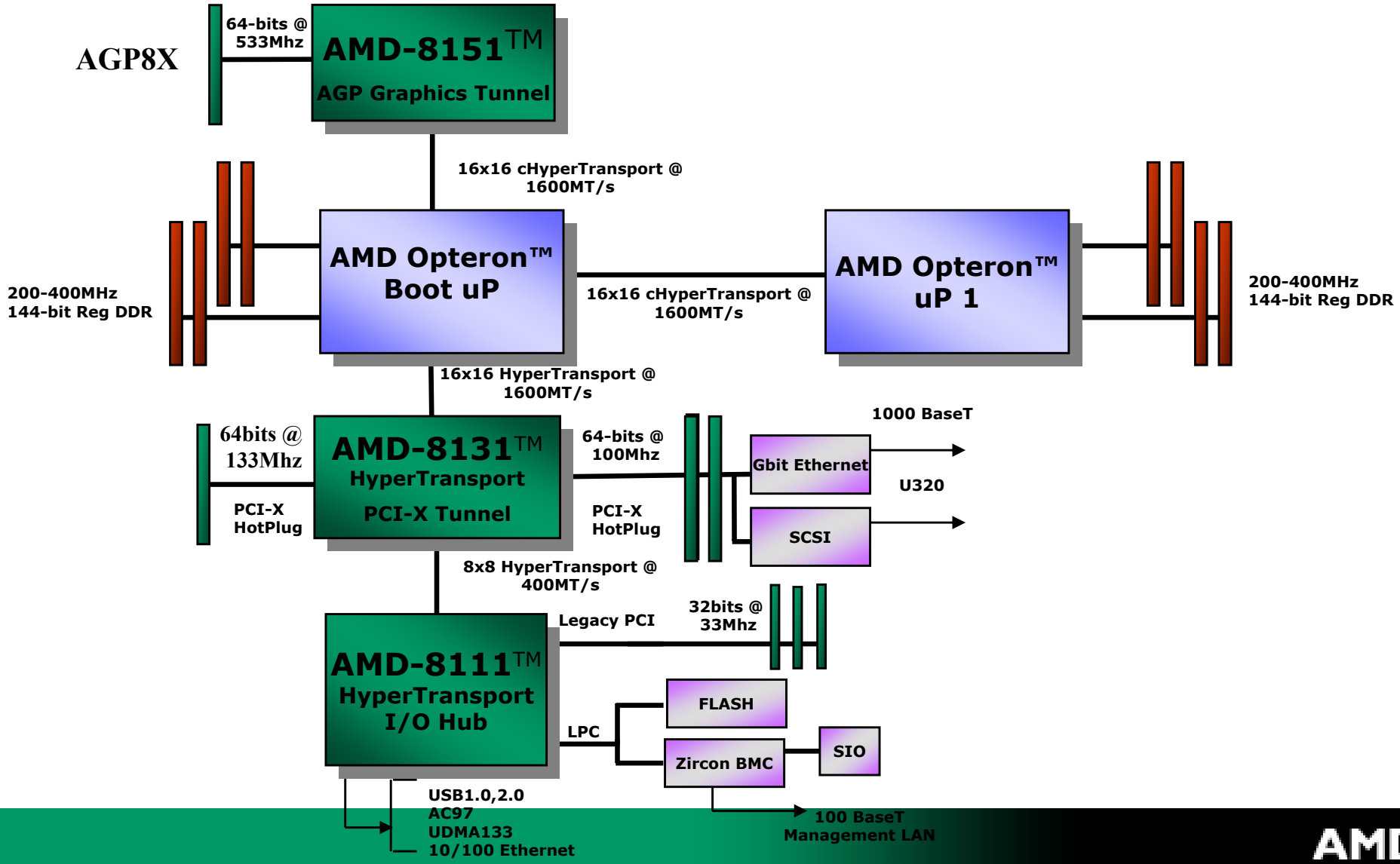
- System scalability limited by northbridge
  - Max of 4 processors
  - Processors compete for FSB bandwidth (1/#proc vs. UP)
  - Electrical issues limit FSB speed vs. UP
  - Memory size and bandwidth are limited
  - Max of 3 PCI-X bridges
  - Many more chips required



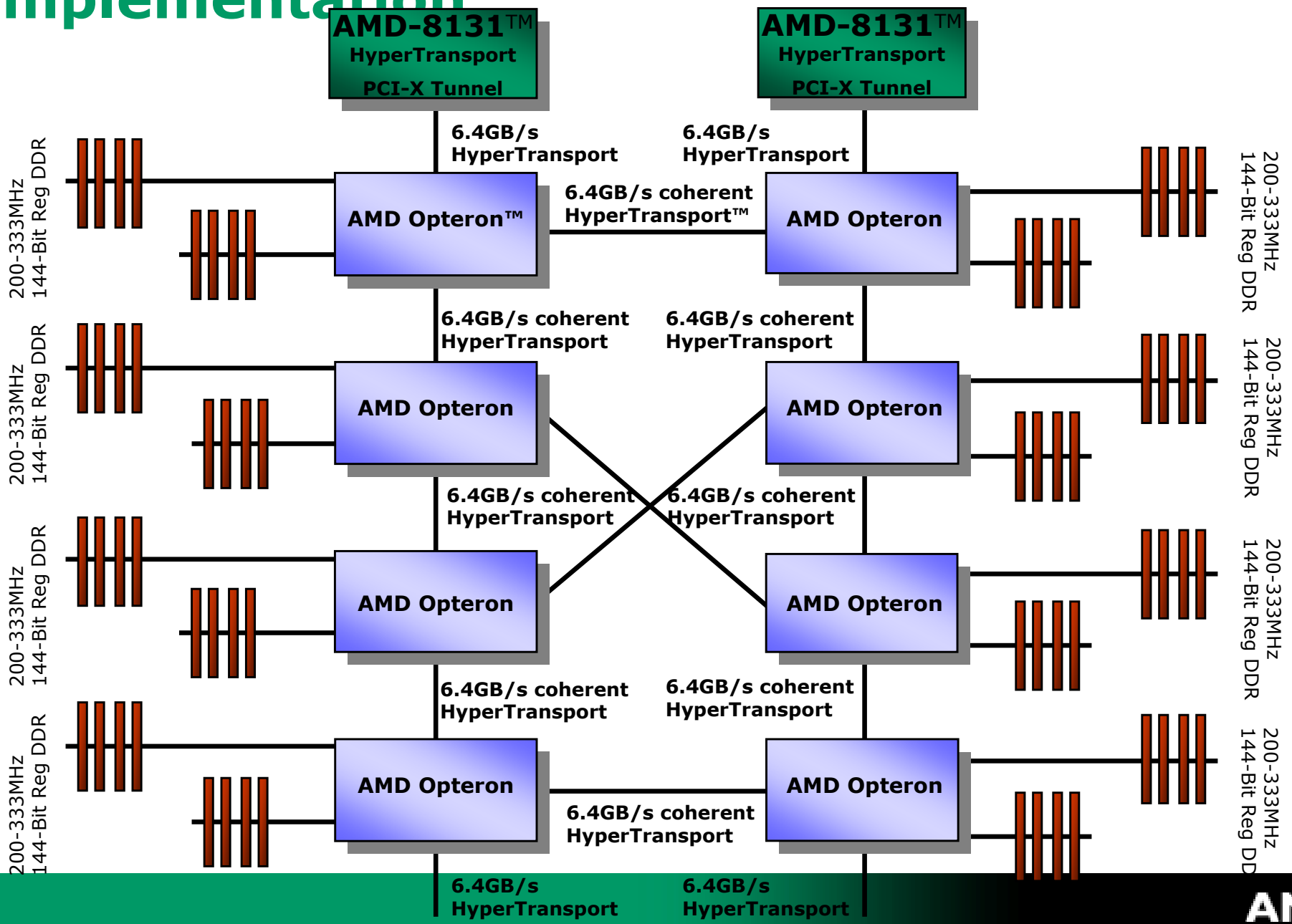
2-Socket Opteron™ Server Implementation



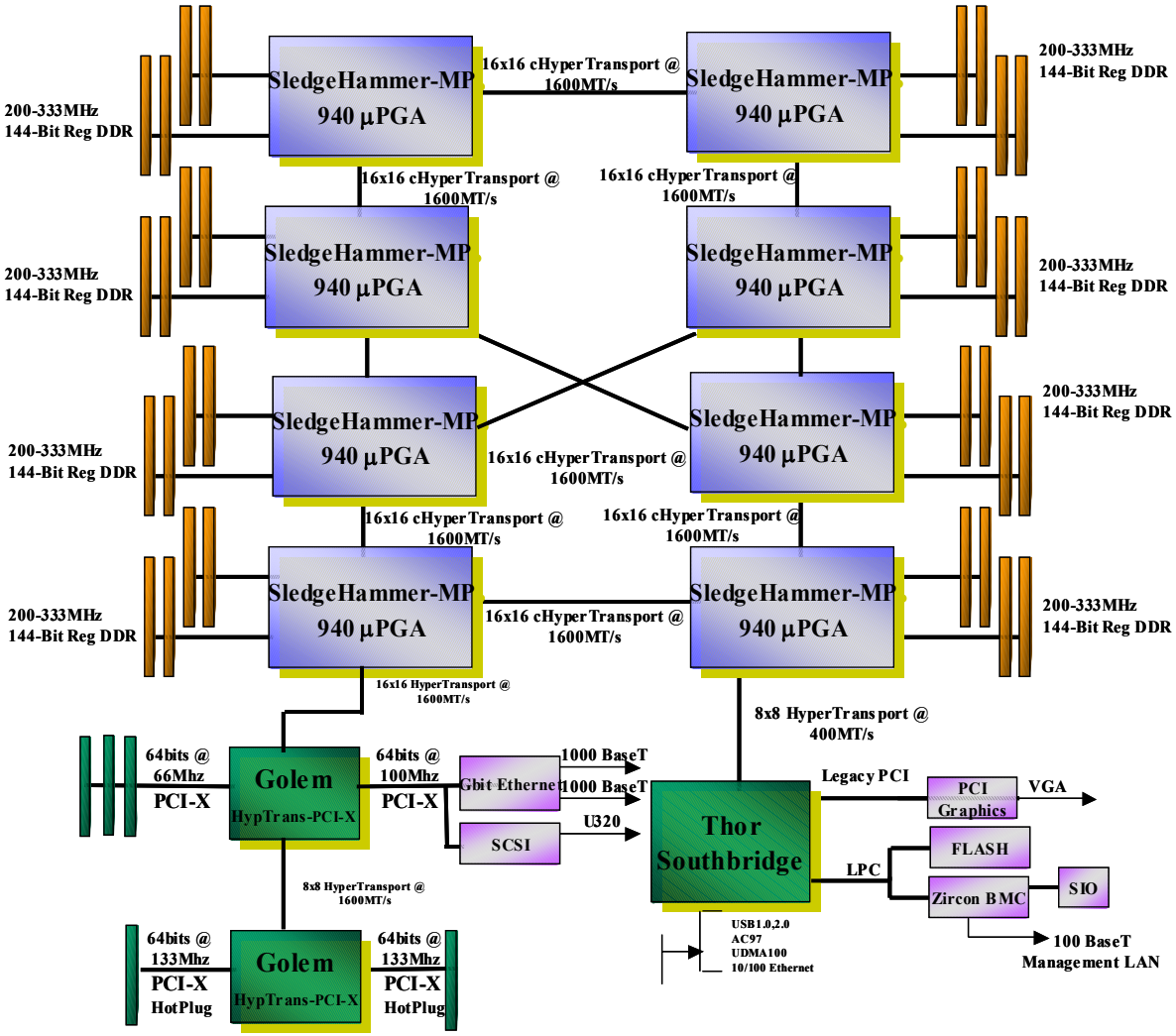
2-Socket AMD Opteron™ Workstation



8-Socket Opteron™ Server Implementation



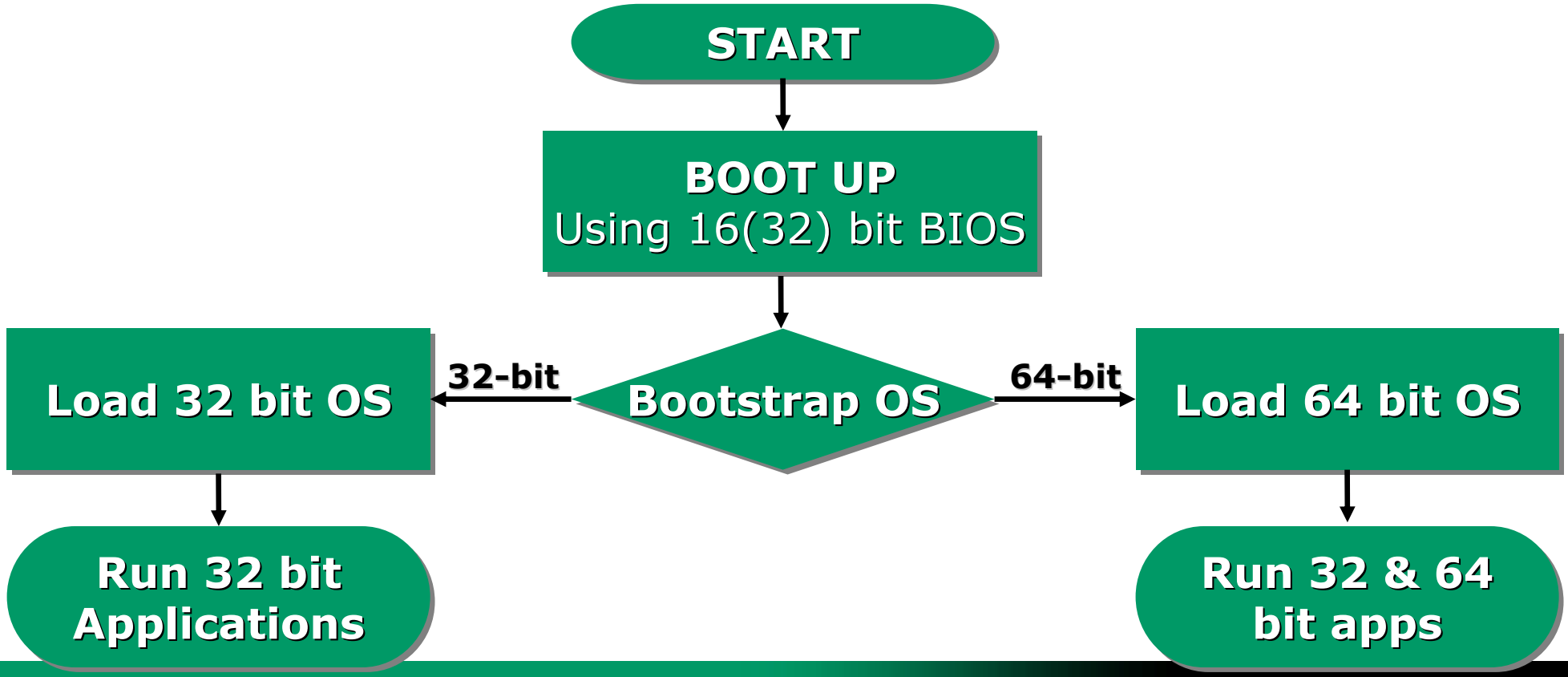
Opteron 8P Server

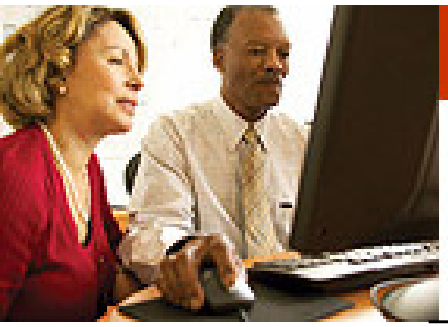


- Demonstration vehicle

AMD64 Technology: Compatibility

- ***An AMD64 PC can run***
- ***Both 32- and 64-bit operating systems***



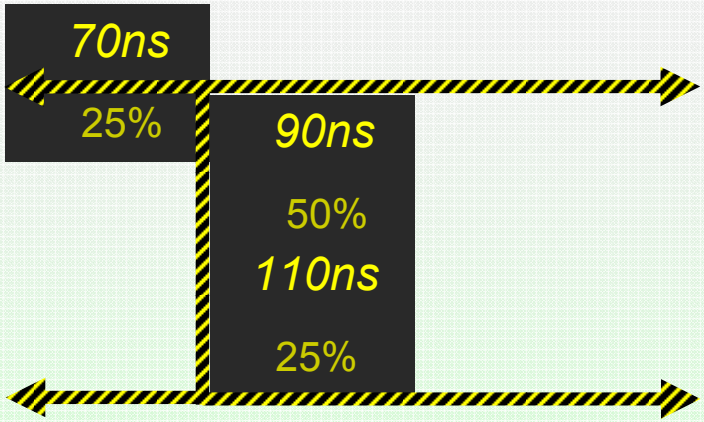


Architectural Comparisons

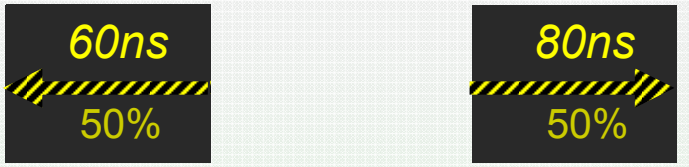
IO Advantages of AMD Dual-Core Memory Latency

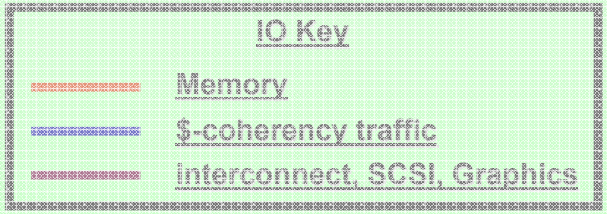
6.4 GB/s

4P Single Core Opteron Platform



4P Dual Core Opteron Platform





$$\langle 4P^{SC} \rangle = \frac{70 + 2 \cdot 90 + 110}{4} = 90\,ns \qquad \langle 4P^{DC} \rangle = \frac{2 \cdot 60 + 2 \cdot 80}{4} =$$

Dual Core Memory latency is substantially better than the Comparable Single Core Equivalent system

Why Multi-Core Processors?

- Multicore is the 20xx way to keep Moore's law (density) continuing to apply to performance
  - Rebalances value equation of performance, power consumption and die area
- Better Performance/Watt than single core
  - Slight drop in core frequency = big power drop
    10% frequency drop results in 40% power drop
- Physics is not kind …
  - Thermal density limits useful scaling of single-core frequency
  - Shrink from 130nm to 90nm
    50% area reduction – same power
  - Power is concentrated at the CPU core(s)
    Adding bigger caches to single-core CPU does not really help
- At 90nm two extremely high performance processor cores fit easily on an affordable die size (100-200mm^2)

Multi-threading is More Mature

- The application base is more ready
  - Media content

    Algorithm level parallelism, value of real-time responsiveness
  - Database and data mining
  - Multi-programmed single users (virus, compression, media, etc)
  - Yet another return of client/server
  - But don't take all this too far, many problems remain stubbornly single threaded
- Multi-threading development tools and run time libraries are more prevalent.
- Note that Amdahl's law still applies; critical sections still bottleneck code.

Some General Observations

- Decomposing a program leaves you with a "serial" piece and a number of sections that can be threaded ("parallel" piece)
- Use Amdahl's Law to estimate benefit of threading
  - Let "S" be percentage of execution time of serial parts of serial-version of program
  - Let "P" be number of threads issued on same number of CPU cores.

- Then **Speed_UP from_threading ~=** $\left[S + \frac{1-S}{P} \right]^{-1}$
- E.G, if program spends 25% of execution time in serial portion of serial version, then Speed UP on a dual-core could be 1.6x
- E.G, if program spends 75% of execution time in serial portion of serial version, then Speed UP on a dual-core could be 1.14x

General Observations (cont)

- Amdahl's law assumes overall structure of program doesn't change going from a serial version to a serial+parallel version
  - If the structure of the serial+parallel-version of the program is drastically different (and more efficient) you may do better than Amdahl's law
- Threading has some small OS/Program overhead that may be noticeable for some programs on non-SMP systems
  - May lead to threaded program running slower than serial version on a non-SMP system
  - Application should test at install time or run-time to determine if the number of available processors allow threading

General Observations (cont)

- Software should use OS APIs to test for number of processors
  - Software may also use CPUID functions to specifically test for Dual-Core capability.
- Threading Performance in Multi-Processor systems also affected by OS Memory Affinity Algorithms
  - May require threads to allocate their own memory at thread initialization rather than relying on parent or the heap.

    Linux for AMD64 assigns memory affinity at first use rather than allocation; other Operating Systems may be different.
  - Not an issue for a single-processor Dual-Core system.

Some References for Multi-Threading

- *Programming with POSIX(R) Threads*, by David R. Butenhof
  - Addison-Wesley Pub Co; ISBN: 0201633922
- *Multithreaded Programming With PThreads*, by Lewis&Berg,
  - Sun Microsystems Press ISBN: 0136807291
- *The Native POSIX Thread Library for Linux* by Drepper & Molnar, January 30, 2003
  - http://people.redhat.com/drepper/nptl-design.pdf

Reason: Physics Is Not Always Kind

- Thermal density limits useful scaling of single-core frequency
  - Trying to work-around these limits breaks the platform infrastructure, increases cost and complexity, reduces yields and reliability.
- Meanwhile, we are able to add ever more transistors as we shrink process dimensions: 90nm, 65nm, 45nm, 32nm
  - But just shrinking from 130nm to 90nm, while reducing die area by 50%, doesn't help change power if you design transistors to MAX Frequency.
- Solution: re-balance the equation of performance, power consumption and die area:
  - Dropping the core frequency slightly can lead to a big power drop
    Transistors designed for < 100% of the MAX Frequency possible, are much more power efficient.
  - Boost performance further (and use the die space) by adding multiple cores.
- E.g. at 90nm two extremely high performance processor cores fit easily on an affordable die size (100-200mm^2)